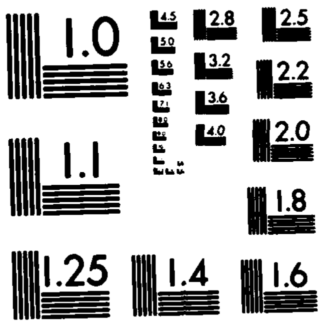
AD-A143 501 ADA (TRADEMARK) TRAINING CURRICULUM PROGRAMMING METHODOLOGY M203 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA JUL 84 DAAB07-83-C-K514 1/8

UNCLASSIFIED F/G 5/9 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

July 1984

Ada® Training Curriculum

Programming Methodology
M203
Teacher's Guide

Center For Tactical Computer Systems
(CENTACS)

U.S. Army Communications-Electronics Command
(CECOM)

Contract DAAB07-83-C-K514

Prepared By:

SOFTECH, INC.
460 Totten Pond Road
Waltham, MA 02154

®Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

PROGRAMMING METHODOLOGY (M303)

INSTRUCTOR NOTES

ALLOCATE 60 MINUTES FOR THIS SECTION. ALLOCATE THE FOLLOWING TIMES FOR THE SUBSECTIONS:

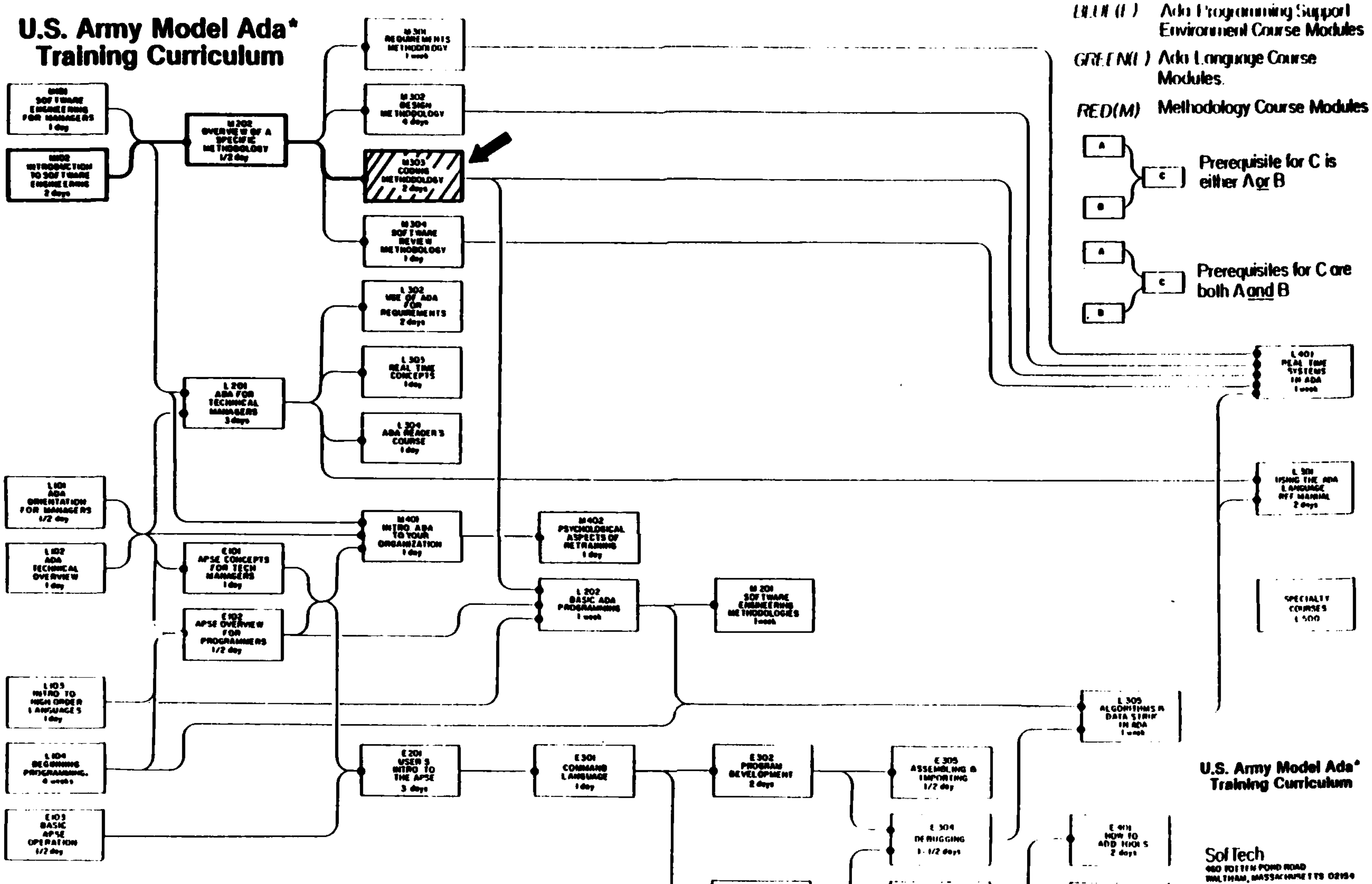
- INTRODUCTIONS (20 MINUTES)
- REVIEW OF THE LIFE-CYCLE (10 MINUTES)
- CODING PHASE (10 MINUTES)
- GOALS OF PROGRAMMING METHODOLOGY (20 MINUTES)

Section 1
INTRODUCTION

VG 817

INSTRUCTOR NOTES

- INTRODUCE YOURSELF AND OTHER INSTRUCTORS (IF APPLICABLE) TO THE CLASS.
- IF CLASS IS OF REASONABLE SIZE (LESS THAN 25), HAVE EACH STUDENT BRIEFLY INTRODUCE THEMSELVES, STATING WHAT THEY HOPE TO GET OUT OF THE COURSE.
- WALK THROUGH THE HIGHLIGHTED ADA CURRICULUM MAP, SHOWING HOW THIS COURSE FITS INTO THE <u>ENTIRE</u> ADA CURRICULUM.
- POINT OUT THAT THIS MODULE IS INDIVISIBLE FROM L202.



*Ada is a trademark of the U.S. Department of Defense (Ada Joint Program Office). The U.S. Army Model Ada Training Curriculum was designed by SofTech, Inc. under the Ada Software Methods Formulation contract (DAAK80-81-C-1087) sponsored by the Software Technology Development Division (CENTACS) of the U.S. Army Communications Electronics Command (CECOM), Ft. Monmouth, N.J.

INSTRUCTOR NOTES

EXPLAIN THAT THIS MODULE HAS 5 SECTIONS.

STATE THE BASIC IDEA OF WHAT IS COVERED IN EACH SECTION. KEEP THIS DISCUSSION CONCEPTUAL.

EXPLAIN THAT THIS SLIDE WILL BE REPEATED AT THE BEGINNING OF EACH SECTION WITH THE APPROPRIATE SECTION HIGHLIGHTED.

OUTLINE

1. **INTRODUCTION**
2. STRUCTURED PROGRAMMING
3. CODING STYLE
4. ENSURING RELIABILITY
5. REVIEW AND WRAP-UP

INSTRUCTOR NOTES

POINT OUT THAT THE <u>PROGRAMMER</u> IS RESPONSIBLE TO PRODUCE CODE THAT IS UNDERSTANDABLE AND CORRECT. THESE ARE NECESSARY FOR CODE TO BE RELIABLE.

MODULE GOAL

- TO DESCRIBE THE PROGRAMMER'S RESPONSIBILITIES DURING THE CODING PHASE

INSTRUCTOR NOTES

POINT OUT THAT THE MODULE COVERS SUCH TECHNIQUES AS:

- STRUCTURED PROGRAMMING CONCEPTS
- STYLE
- FORMATTING
- LOOP INVARIANTS
- ETC.

MODULE GOAL

- TO TEACH MODERN CODING TECHNIQUES

INSTRUCTOR NOTES

NO PROOFS WILL BE GIVEN, BUT RATHER PRACTICAL USES OF THE RESULTS WILL BE EXPLAINED.

MODULE GOAL

- TO PROVIDE THE BACKGROUND FOR THE PRACTICAL USE OF THESE TECHNIQUES

INSTRUCTOR NOTES

TALK ABOUT THE GENERAL THRUST OF THE MODULE IN TERMS OF WHAT WILL BE COVERED. WHEN TALKING ABOUT THESE TOPICS RELATE THE TOPIC BACK TO ONE (OR MORE) OF THE THREE MODULE GOALS. EMPHASIZE THAT THIS MODULE DOES NOT ATTEMPT TO TEACH THE ADA LANGUAGE.

REMAINDER OF COURSE

1. INTRODUCTION
   - REVIEW OF SOFTWARE LIFE CYCLE
   - CODING PHASE
   - GOALS

2. STRUCTURED PROGRAMMING
   - CONTROL STRUCTURES
   - WHY'S AND WHEREFORES

3. CODING STYLE
   - FORMATTING CONVENTIONS
   - COMMENTING CONVENTIONS
   - NAMING CONVENTIONS

4. ENSURING RELIABILITY
   - CORRECTNESS
   - DOCUMENTATION
   - UNIT TESTING

5. REVIEW AND WRAP UP

INSTRUCTOR NOTES

DISCUSS THE TIME MAP.

POINT OUT THE EXERCISES. TELL THE CLASS WHAT THEY WILL DO AS EXERCISES.

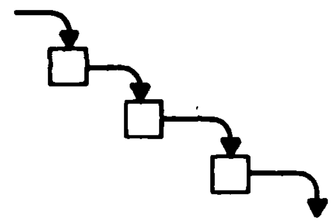
PROPOSED SCHEDULE

DAY 1	DAY 2
INTRODUCTION	CODING STYLE
BREAK	BREAK
STRUCTURED PROGRAMMING DEFINITIONS & MOTIVATIONS	RELIABLE PROGRAMMING TECHNIQUES
LUNCH	LUNCH
STRUCTURED PROGRAMMING METHODS, COSTS & BENEFITS	RELIABLE PROGRAMMING DOCUMENTATION UNIT TESTING
BREAK	BREAK
STRUCTURED PROGRAMMING EXERCISE	REVIEW AND WRAP-UP EXERCISE

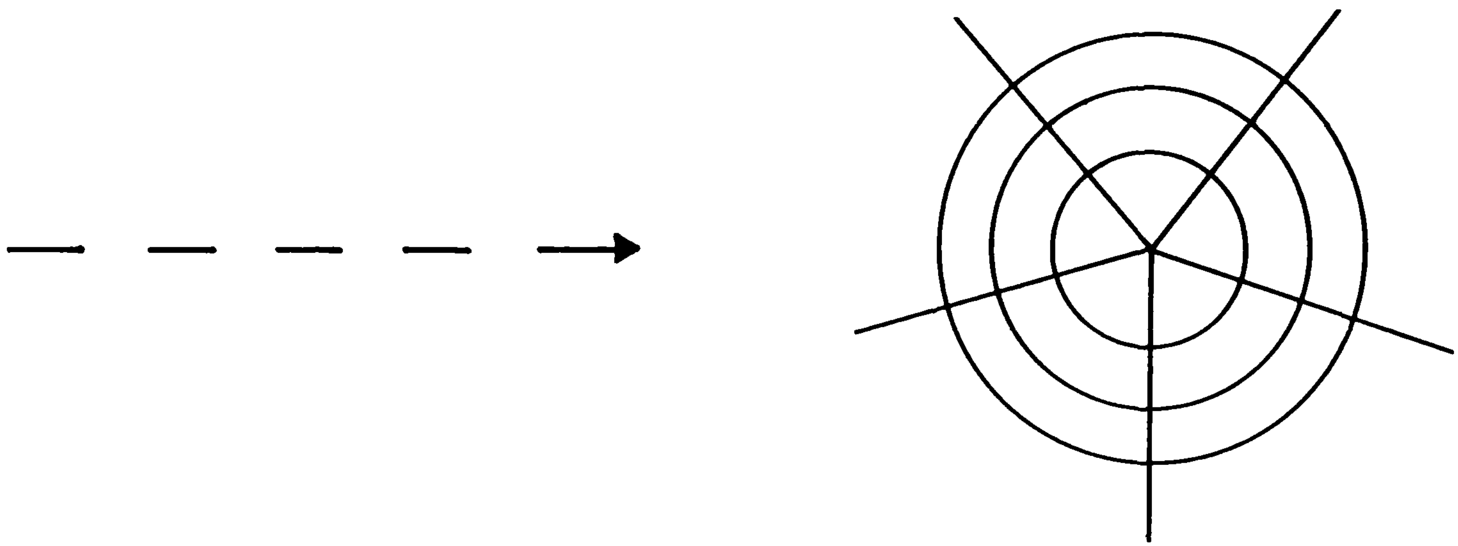
INSTRUCTOR NOTES

THE MAIN MESSAGE OF THIS SECTION IS TO PROVIDE AN OVERVIEW OF THE ENTIRE SOFTWARE DEVELOPMENT PROCESS SO THAT IT IS CLEAR HOW THE PROGRAMMING PHASE FITS IN.

THE STUDENT NEEDS A GOOD UNDERSTANDING OF THE PHASES IN THE LIFE CYCLE AS WELL AS THE INPUTS AND OUTPUTS OF EACH PHASE.



LIFE CYCLE



INSTRUCTOR NOTES

DO NOT GET BOGGED DOWN IN DISCUSSION OF DIFFERENT ORGANIZATIONS VIEW OF THE LIFE CYCLE. POINT OUT THAT THIS SECTION IS JUST INTENDED TO GIVE VARIOUS GLOBAL VIEWS OF THE PROCESS SO THE STUDENT HAS A CONCEPT OF WHERE HE/SHE FITS IN THE PROCESS.

THE IMPORTANT POINT HERE IS THE FACT THAT IT IS NON-STANDARD.

SOFTWARE DEVELOPMENT LIFE CYCLE

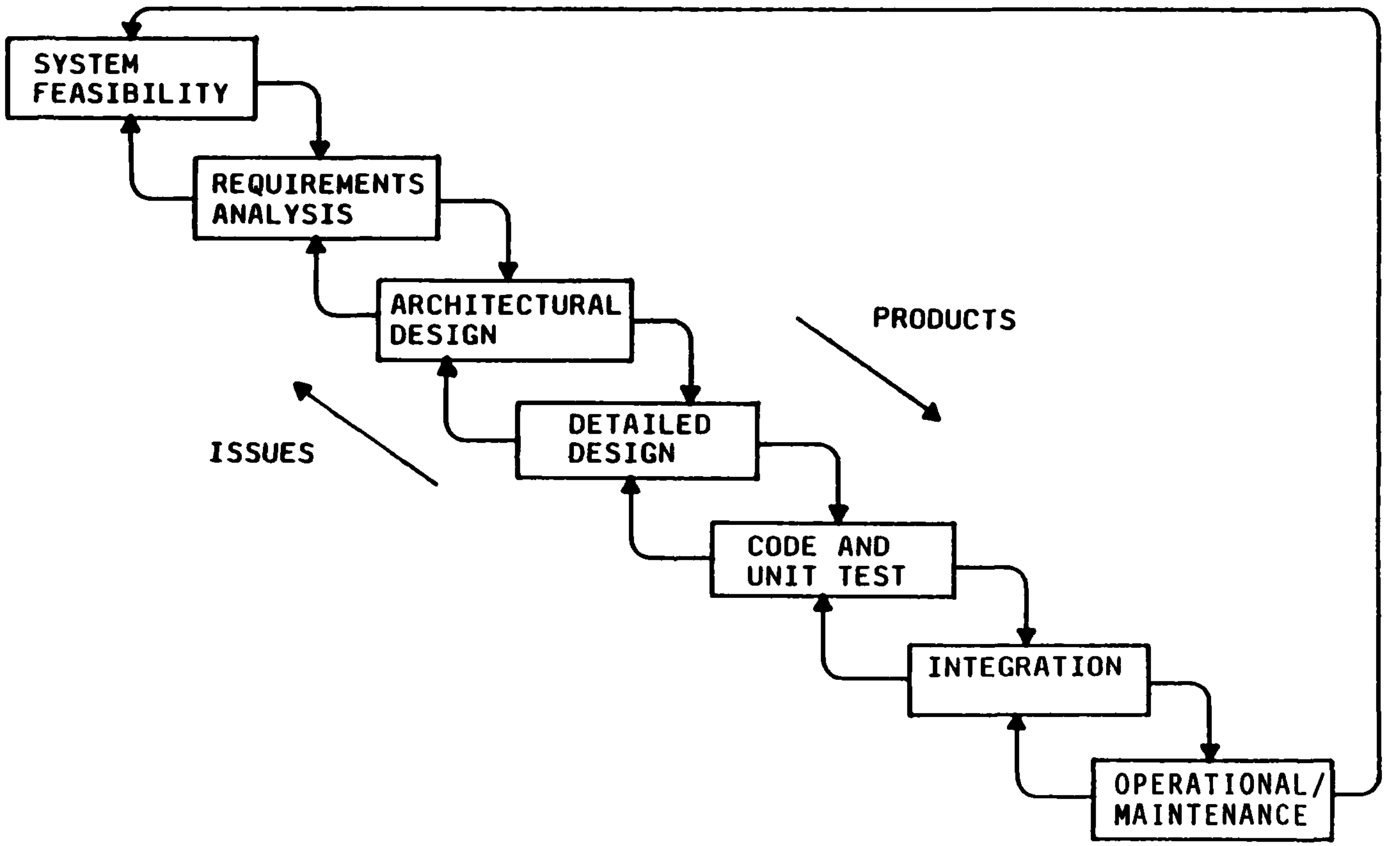
- IS NOT STANDARD THROUGHOUT THE INDUSTRY
- SEEMS TO BE CONVERGING TOWARDS A STANDARD
- PROVIDES A STRUCTURE FOR THE MEASUREMENT AND CONTROL OF THE DEVELOPMENT PROCESS

	FEASIBILITY	REQUIREMENTS ANALYSIS	ARCHITECTURAL DESIGN	DETAILED DESIGN	CODE/ UNIT TEST	INTEGRATION	OPERATIONAL/ MAINTENANCE
PRIMARY FUNCTION	DETERMINE THAT IT CAN BE DONE. DETERMINE THE USERS' NEEDS.	DETERMINE WHAT THE PRODUCT WILL DO.	DETERMINE HOW (GENERALLY) THE SYSTEM WILL FUNC- TION.	DETERMINE HOW (SPECIFICALLY) THE SYSTEM WILL WORK.	CODE MODULES AND DEBUG.	TEST THE SYS- TEM TO BE SURE IT WORKS.	INSTALL THE SYSTEM FOR CUSTOMER AND FIX/ENHANCE SYSTEM.
ADDITIONAL ACTIVITIES	SELECT TOP-LEVEL PERSONNEL.	FORM REQTS FOR ACCEP- TANCE TESTS. FORM TOP- LEVEL TEST PLANS. DOCUMENT RE- QUIREMENTS. OUTLINE USER MANUAL.	SELECT PERSONNEL. ACQUIRE SUP- PORT TOOLS. UPDATE RE- QUIREMENTS. DRAFT TEST PLANS. DRAFT USER MANUALS. DOCUMENT ARCHITECTURAL DESIGN.	DOCUMENT DE- TAILED DESIGN. FORM DETAILED TEST PLANS. FULL DRAFT USER MANUALS. UPDATE ARCHI- TECTURAL DE- SIGN, REQUIRE- MENTS.	DOCUMENT EACH COMPONENT. FORM INTEGRA- TION PLANS. UPDATE DESIGN, REQUIREMENTS.	FINAL MANUALS UPDATE CODE, DESIGN, RE- QUIREMENTS.	CONTINUING UPDATE OF DOCUMENTS, SOFTWARE, DESIGN.
PRODUCTS	PLANS	REQTS	TOOLS ARCH DESIGN	DETAILED DESIGN	CODE AND UNIT TESTS	INTG. TESTS	SYSTEM AND CHANGES

DISCUSS HOW THE PHASES ARE INTERWOVEN WITH PREVIOUS AND SUCCESSIVE PHASES.

DISCUSS HOW EACH SUCCESSIVE PHASE REFINES UNDERSTANDING.

GENERALIZED MODEL



INSTRUCTOR NOTES

STRESS THAT THE NEED IS PERCEIVED.

B-SPEC IS A STATEMENT OF FUNCTIONALITY. C-SPEC CONTAINS TOP LEVEL ARCHITECTURAL DESIGN (PART I), MODULE STRUCTURE (PART II), AND CODE (PART III).

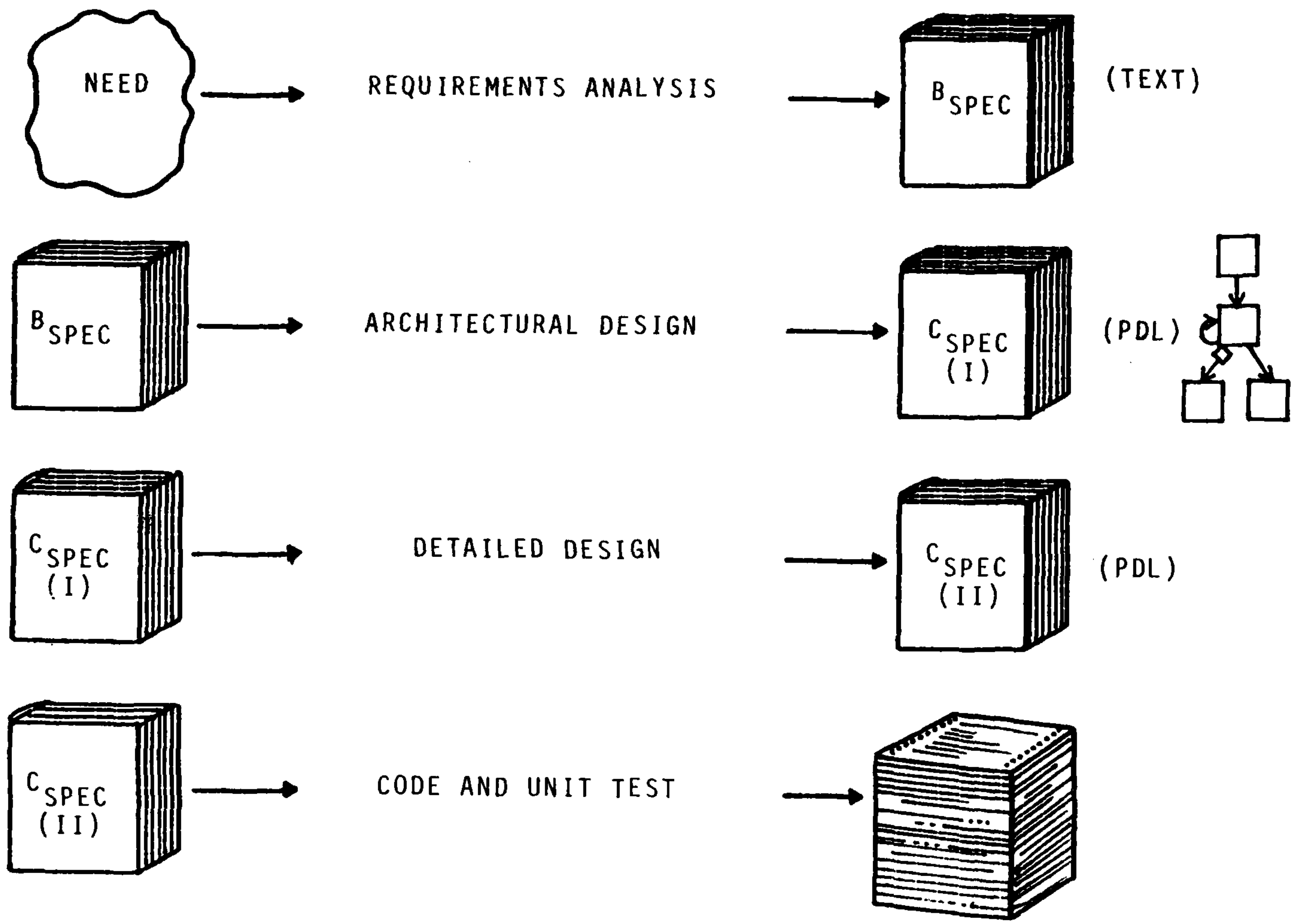
THE TOP LEVEL ARCHITECTURE MAY APPEAR IN VARIOUS FORMS. (PDL, DATA DICTIONARY, DATA FLOW DIAGRAM, ETC.)

DETAILED DESIGN IS THE ASSIGNMENT TO SPECIFIC MODULES.

TEST PLANS ARE A NEBULOUS AREA.

POINT OUT THAT THIS MODULE STRESSES ACTIVITY DURING CODE AND UNIT TEST PHASE.

ANOTHER VIEW



INSTRUCTOR NOTES

THE ANSWER IS AN UNEQUIVOCAL "YES"!

$56M UNIVAC CONTRACT FOR UNITED RESERVATION SYSTEM AND $217M ADVANCED LOGISTIC SYSTEM CANCELLED AFTER PARTIAL IMPLEMENTATIONS DUE TO INCOMPLETE FEASIBILITY STUDIES.

LARGE PROJECTS NEED TECHNIQUES TO STRUCTURE AND CONTROL THE SYSTEM INTO MANAGEABLE CHUNKS.

IS ALL THIS NECESSARY?

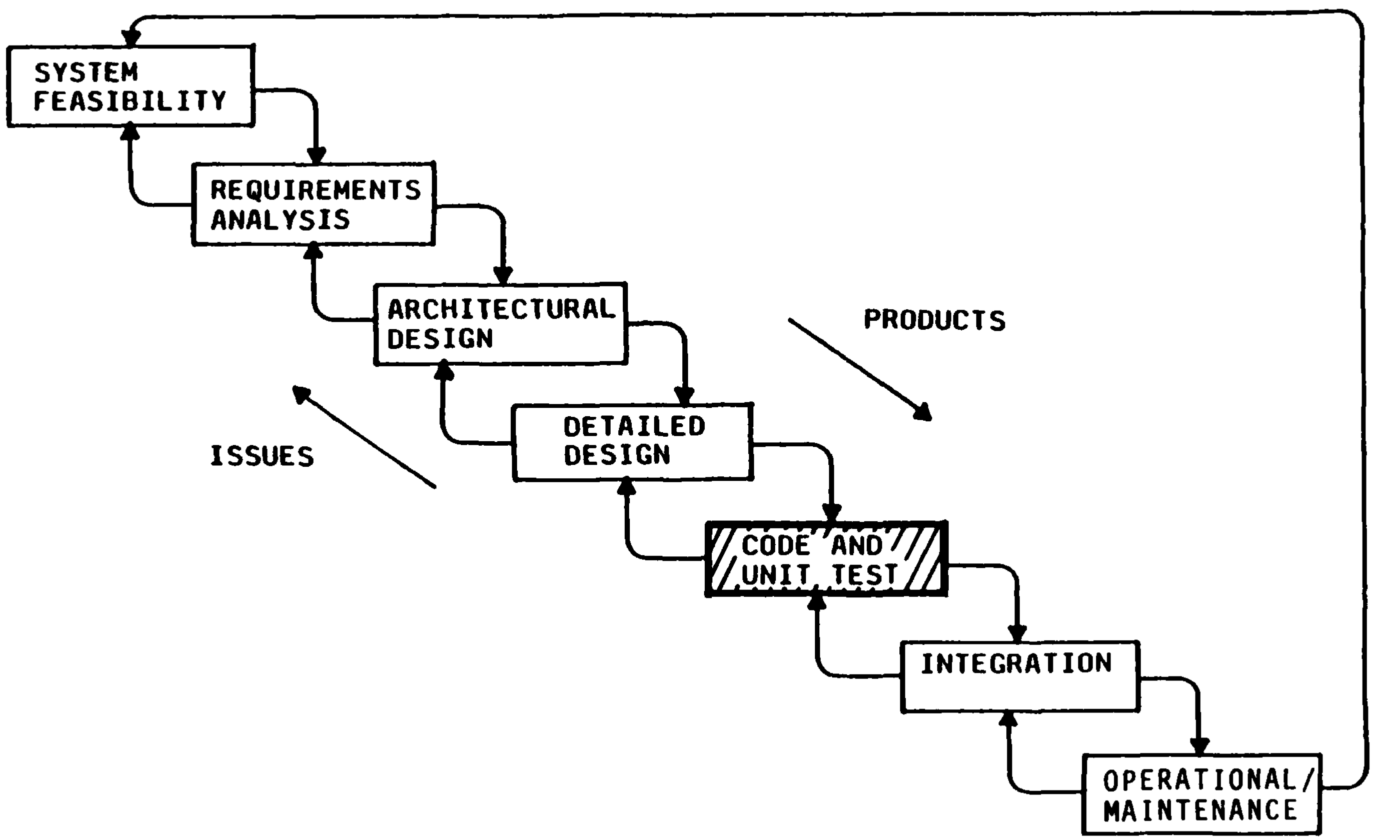
VG 817

INSTRUCTOR NOTES

RETURNING TO OUR GENERALIZED VIEW, THE REMAINDER OF THIS MODULE DEALS WITH THE CODING PHASE.

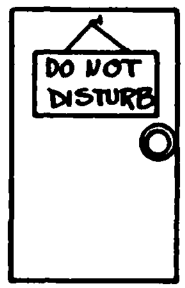
THE STUDENT SHOULD BE REMINDED THAT THIS IS ONLY ONE PHASE. THE POINT OF REVIEWING THE LIFE CYCLE IS TO ENSURE THE STUDENT GETS AN INTUITIVE FEEL FOR WHERE HIS WORK FITS IN THE OVERALL VIEW.

GENERALIZED MODEL



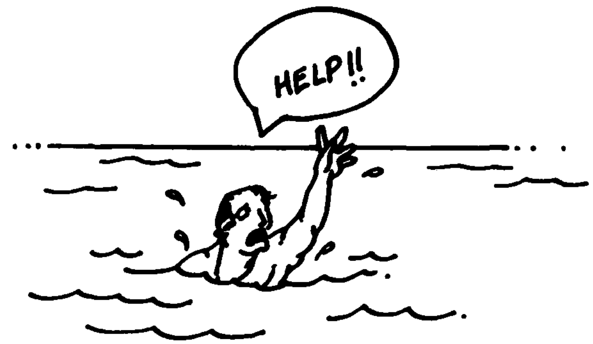
INSTRUCTOR NOTES

THIS IS THE SECOND TOPIC OF THIS SECTION. THIS SUBSECTION DESCRIBES THE CODING PROCESS.




```
for Counter in 1 .. 6
loop
   ...
end loop;
```

CODING PHASE



INSTRUCTOR NOTES

THE POINT IS THAT UNIT TESTING IS AN <u>INTEGRAL</u> PART OF CODING.

CODING PHASE

CODING INCLUDES UNIT TESTING

INSTRUCTOR NOTES

- S/W ARCHITECTURE
- DETAILED DESIGN DOCUMENT
  - MODULE STRUCTURE IN TERMS OF INPUT, PROCESSING, AND OUTPUT
- IDS
  - INTERFACE DESIGN SPECIFICATION
- DB
  - DATABASE DOCUMENT
- PROJECT SPECIFIC DOCUMENTS
  - E.G. DESCRIPTION OF MESSAGE FORMATS
- STANDARDS
  - MILITARY STANDARDS AND/OR PROJECT STANDARDS
- TEST PLANS
  - SYSTEM TEST PLANS

INPUTS TO THE CODING PHASE

- S/W ARCHITECTURE
- DETAILED DESIGN DOCUMENT
- IDS
- DB
- PROJECT SPECIFIC DOCUMENTS
- STANDARDS
- TEST PLANS

INSTRUCTOR NOTES

THE WORD <u>CORRECT</u> WILL BE DEFINED IN THIS MODULE.

OUTPUTS OF THE CODING PHASE

- CORRECT CODE
- TESTED CODE
- UNIT TEST PLAN
- UNIT TEST RESULTS
- UNIT TEST DATA

INSTRUCTOR NOTES

ASK THE QUESTION "WHAT IS RELIABILITY?" LET A COUPLE OF STUDENTS ANSWER THE QUESTION, THEN PROCEED ...

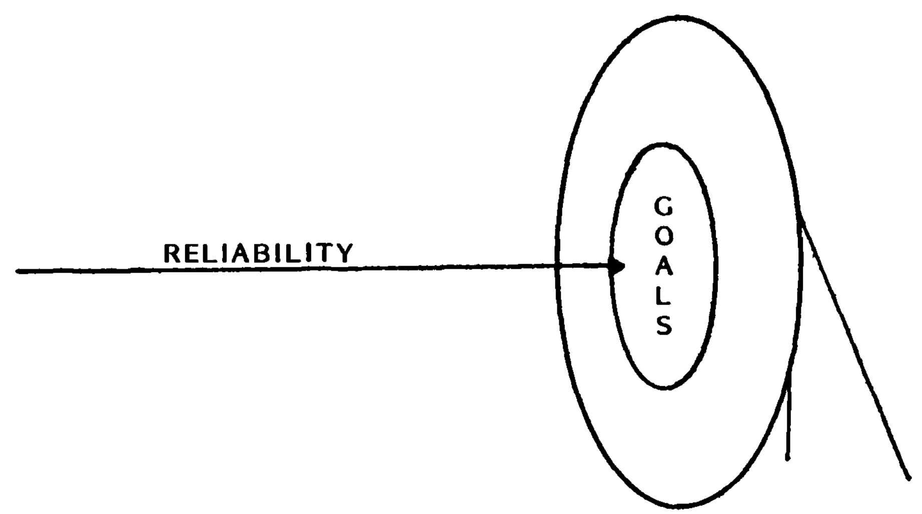
IF A STUDENT DOES NOT VOLUNTEER, JUST PRESS ON. BUT THIS IS A GOOD PLACE FOR A <u>SMALL</u> DISCUSSION.

PROGRAMMER RESPONSIBILITY DURING CODING PHASE

THE <u>PROGRAMMER</u> IS <u>RESPONSIBLE</u> FOR <u>ENSURING</u> PROGRAM <u>RELIABILITY</u>

INSTRUCTOR NOTES

THIS IS AN IMPORTANT SUBSECTION. TAKE MORE TIME HERE THAN IN THE PREVIOUS SUBSECTIONS.


RELIABILITY
GOALS

INSTRUCTOR NOTES

USER SATISFACTION IS ONE ASPECT WHICH IS IMPACTED BY RELIABILITY.

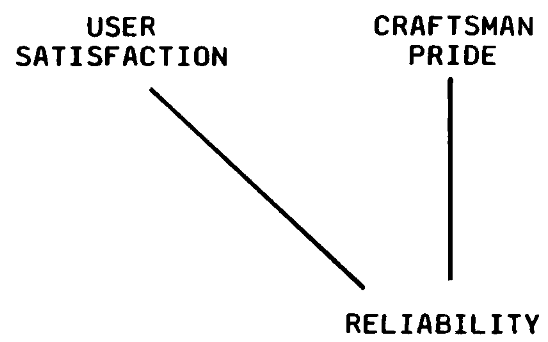
GOALS OF ...

USER
SATISFACTION

RELIABILITY

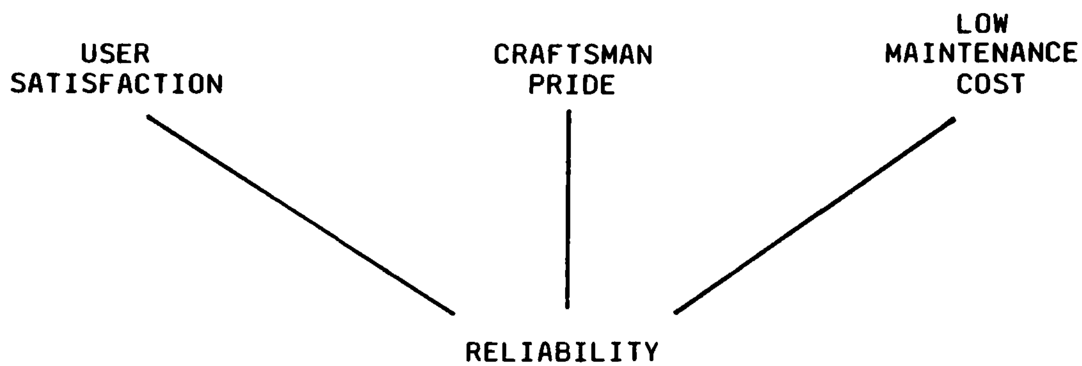
PRIDE IS AFFECTED BY RELIABILITY.

GOALS OF ...



RELIABLE CODE HELPS CONTRIBUTE TO LOWER MAINTENANCE COSTS.

GOALS OF ...

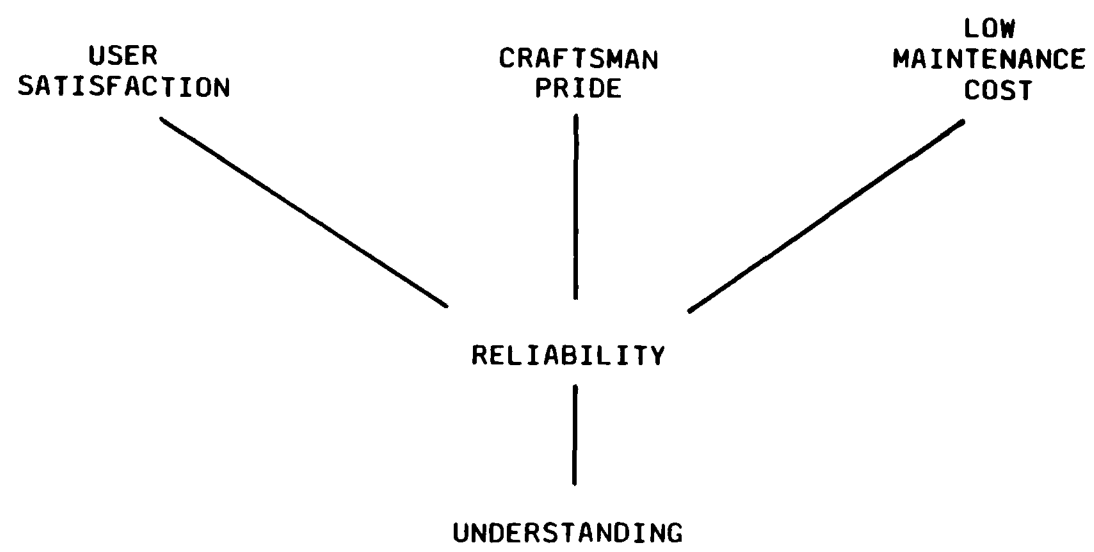

USER SATISFACTION
CRAFTSMAN PRIDE
LOW MAINTENANCE COST
RELIABILITY

INSTRUCTOR NOTES

IF WE DON'T UNDERSTAND OUR OWN CODE HOW CAN WE EXPECT THE CODE TO BE RELIABLE?

GOALS OF ...

ACHIEVING RELIABILITY REQUIRES UNDERSTANDING



INSTRUCTOR NOTES

WHY ISN'T TESTING CONSIDERED THE CORNERSTONE OF RELIABILITY?

TESTING?

INSTRUCTOR NOTES

IT IS IMPOSSIBLE TO TEST EVERY VALUE FOR EVERY PATH.

TESTING

E. DJIKSTRA

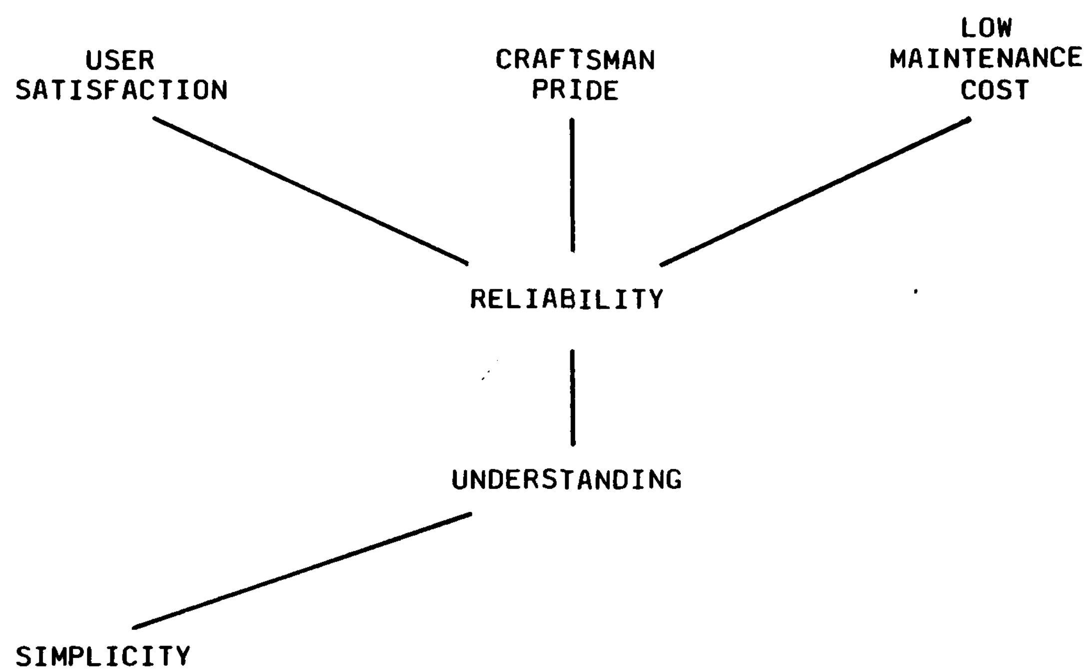
> TESTING ONLY SHOWS THE PRESENCE OF ERRORS, NOT THEIR ABSENCE.

INSTRUCTOR NOTES

K.I.S.S. IS A PRETTY GOOD MAXIM.

GOALS OF ...

SIMPLICITY ACHIEVES UNDERSTANDING



INSTRUCTOR NOTES

GO TO FREE PROGRAMMING

- EACH CONTROL STRUCTURE HAS ONE INPUT AND ONE OUTPUT.

THIS MEANS THAT ANY COMPLICATED PROGRAM BUILT FROM THESE CONTROL STRUCTURES CAN BE CONSIDERED TO BE A SINGLE ACTION.

A PROGRAM ONCE UNDERSTOOD CAN BE CONSIDERED TO BE A SINGLE ACTION AND USED TO UNDERSTAND MORE COMPLICATED PROGRAMS.

SIMPLICITY

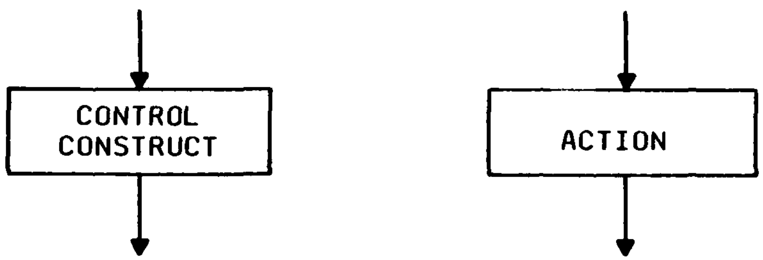
- SIMPLE UNDERSTANDABLE CONTROL STRUCTURES ENABLE US TO UNDERSTAND MORE COMPLEX STRUCTURES

- BY RESTRICTING OURSELVES TO PROGRAMMING WITH A LIMITED SET OF SIMPLE UNDERSTANDABLE CONTROL STRUCTURES, WE ELIMINATE THE NEED FOR goto STATEMENTS

INSTRUCTOR NOTES

TALK ABOUT HOW SINGLE ENTRY-SINGLE EXIT SUPPORTS ABSTRACTION.

SINGLE ENTRY - SINGLE EXIT



INSTRUCTOR NOTES

ANOTHER WAY OF VIEWING SIMPLICITY.

SIMPLICITY

THE CONDITIONS UNDER WHICH VARIOUS SECTIONS OF CODE ARE EXECUTED ARE EXPLICIT IN THE BOOLEAN EXPRESSION CONTROLLING CONDITIONAL AND ITERATIVE STATEMENTS.

INSTRUCTOR NOTES

ENUMERATION FOR SEQUENTIAL AND CONDITIONAL STRUCTURES. INDUCTION FOR ITERATION.

THE MAIN POINT IS THAT THERE ARE MATHEMATICAL TOOLS THAT ARE APPLICABLE IN THE WORLD OF STRUCTURED PROGRAMMING AND HELP PRODUCE "SIMPLE" CODE.

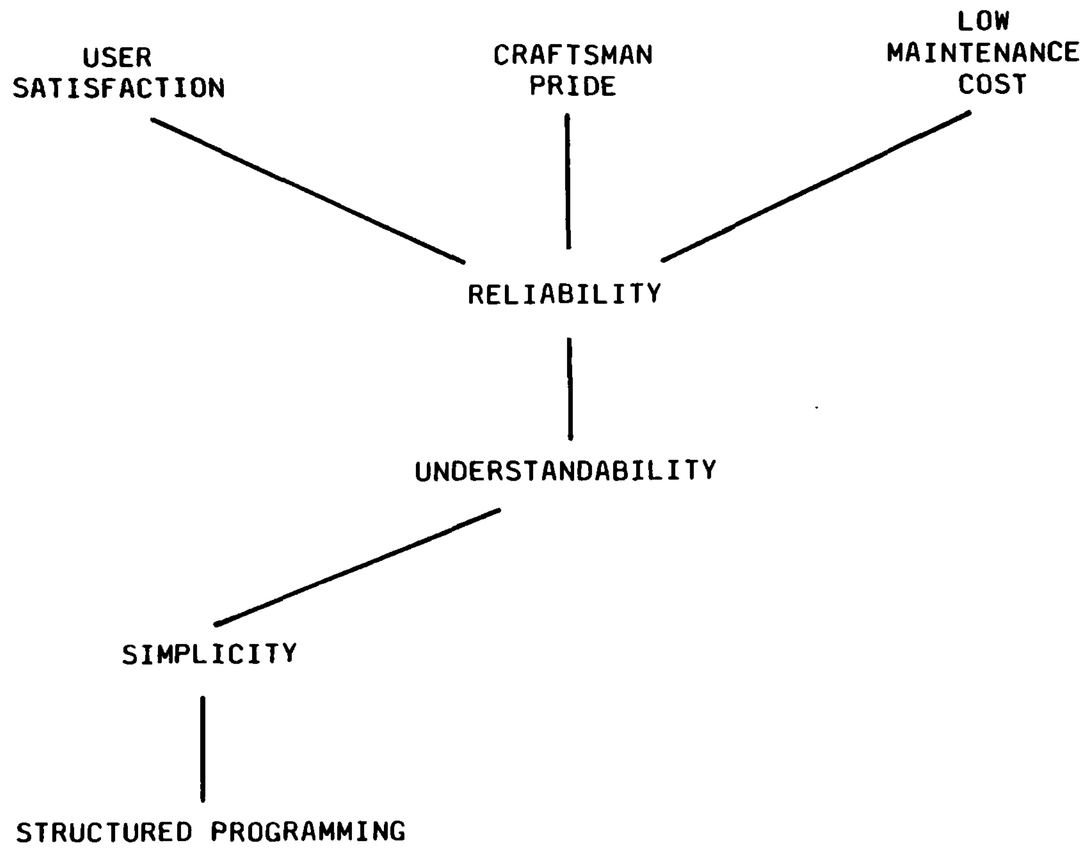
SIMPLICITY

WE HAVE MATHEMATICAL TOOLS FOR DEALING WITH THE STRUCTURED PROGRAMMING OPERATIONS.

INSTRUCTOR NOTES

STRUCTURED PROGRAMMING, OTHERWISE KNOWN AS "GO TO FREE" PROGRAMMING IS A METHODOLOGY TO BE USED IN CREATING "SIMPLER" CODE.

GOALS OF ...

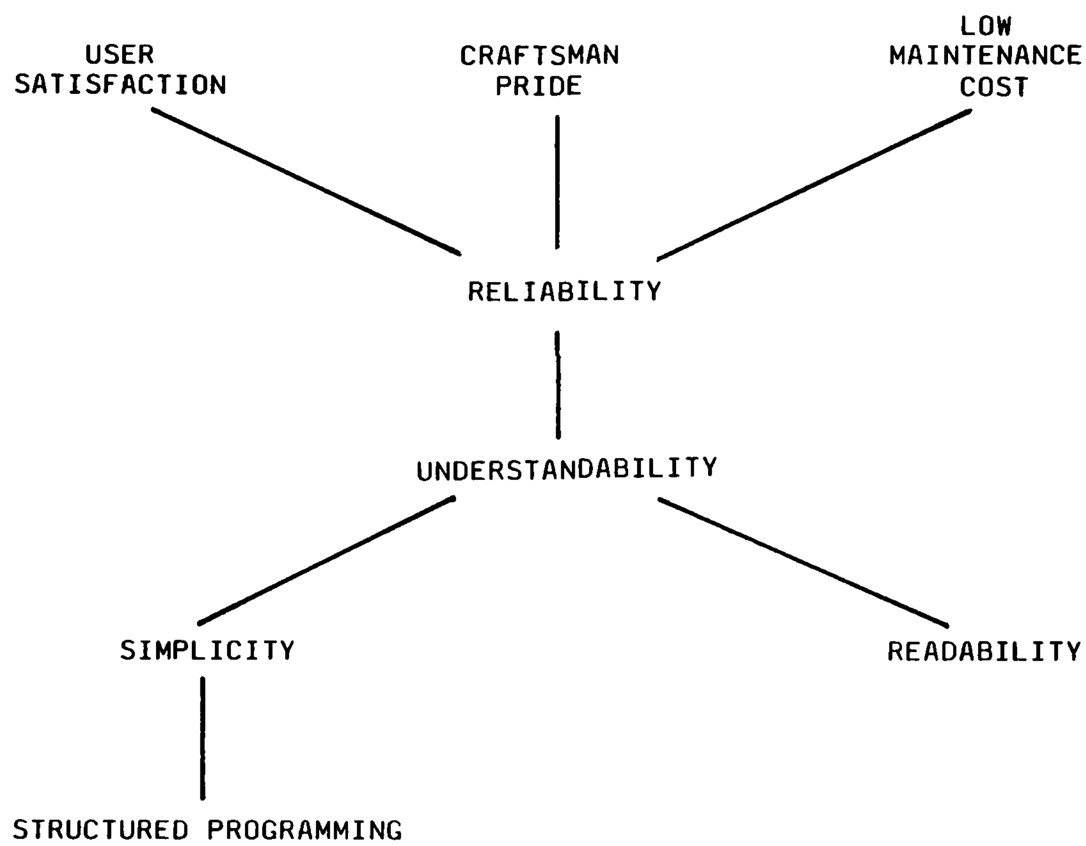

USER SATISFACTION
CRAFTSMAN PRIDE
LOW MAINTENANCE COST
RELIABILITY
UNDERSTANDABILITY
SIMPLICITY
STRUCTURED PROGRAMMING

INSTRUCTOR NOTES

$$\left.\begin{array}{l}\texttt{Lowest\_Value}\\ \texttt{Highest\_Value}\end{array}\right\}\ \text{ARE EASIER TO READ THAN}\ \left\{\begin{array}{l}\texttt{X}\\ \texttt{Y}\end{array}\right.$$

GOALS OF ...

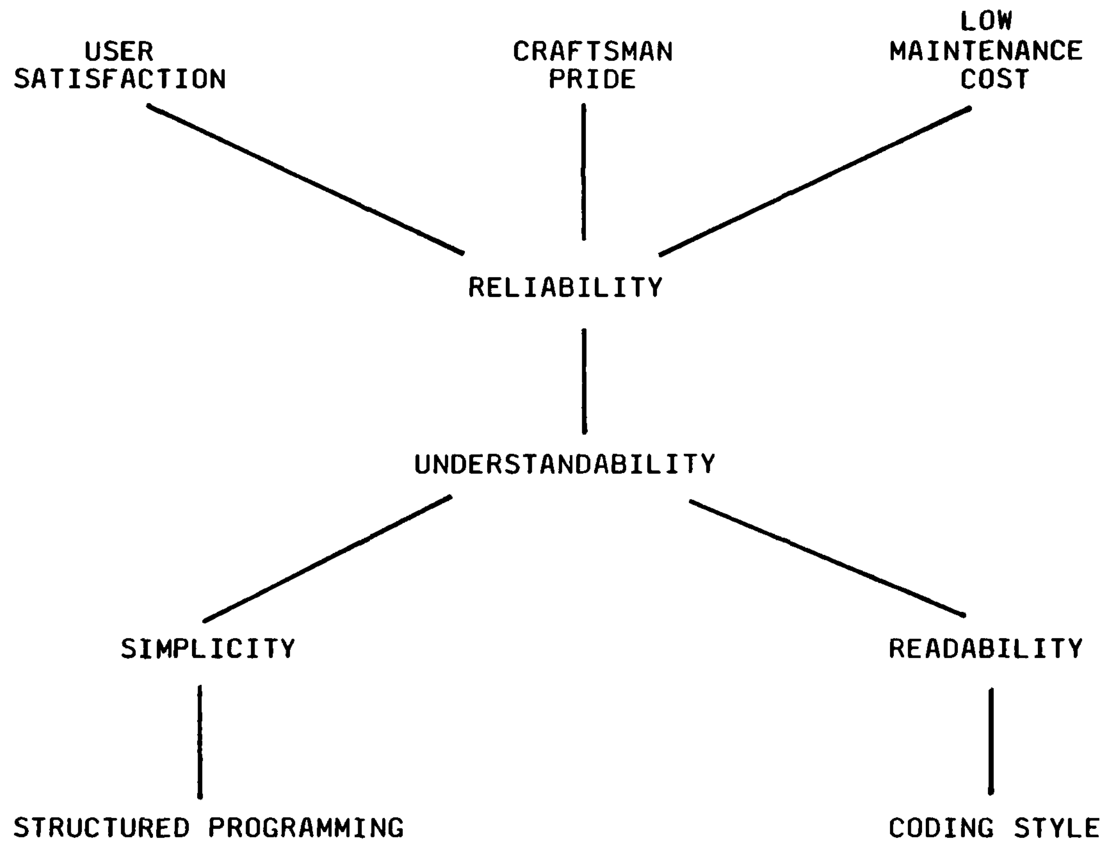
READABLE CODE IS EASIER TO UNDERSTAND.



INSTRUCTOR NOTES

JUST AS WE HAVE A METHODOLOGY TO SIMPLIFY THE CODE, WE HAVE A METHODOLOGY TO HELP CREATE MORE READABLE CODE.

GOALS OF ...


USER SATISFACTION
CRAFTSMAN PRIDE
LOW MAINTENANCE COST
RELIABILITY
UNDERSTANDABILITY
SIMPLICITY
READABILITY
STRUCTURED PROGRAMMING
CODING STYLE

WE MUST USE PROGRAMMING METHODOLOGY TO ENSURE THAT OUR PROGRAMS ARE UNDERSTANDABLE.

BREAK

SUMMARY

- PROGRAMMER'S RESPONSIBILITY IS TO PRODUCE RELIABLE CODE
- TO DO SO HE/SHE MUST UNDERSTAND THE CODE
- TO BE UNDERSTANDABLE, CODE MUST BE SIMPLE AND READABLE
- USE STRUCTURED PROGRAMMING TO ACHIEVE SIMPLE CODE
- USE GOOD CODING STYLE TO ACHIEVE READABLE CODE

INSTRUCTOR NOTES

Section 2
STRUCTURED PROGRAMMING

VG 817

INSTRUCTOR NOTES

THE REST OF THE DAY WILL BE SPENT ON STRUCTURED PROGRAMMING.

OUTLINE

1. INTRODUCTION

2. **STRUCTURED PROGRAMMING**

3. CODING STYLE

4. ENSURING RELIABILITY

5. REVIEW AND WRAP-UP

INSTRUCTOR NOTES

STRUCTURED PROGRAMMING IS REALLY A METHODOLOGY AS IT EMBODIES CONCEPTS, RULES, ETC.

DEFINITION

THERE IS NO "STANDARD" DEFINITION.

INSTRUCTOR NOTES

MUCH OF THE LITERATURE USES THIS NOMENCLATURE.

DEFINITION

YOU MAY HAVE HEARD OF IT AS

"GO TO FREE PROGRAMMING"

BUT IT REALLY IS "... NOT THE ABSENCE OF GOTO'S ..." BUT THE "... PRESENCE OF STRUCTURE ..."

MILLS, H.D. MATHEMATICAL FOUNDATIONS FOR STRUCTURED PROGRAMMING FSC 72-6112, IBM, FEBRUARY 1972.

INSTRUCTOR NOTES

POINT IS THAT CLEAR THINKING CAN HELP US CHANGE PROGRAMMING FROM A FRUSTRATING TRIAL AND ERROR ACTIVITY TO A SYSTEMATIC QUALITY CONTROLLED ACTIVITY.

DIJKSTRA FELT THAT THE BEST APPROACH TO PROVING A PROGRAM CORRECT IS TO CONTROL ITS STRUCTURE.

FIRST NAME

EDSGER DIJKSTRA ORIGINATED A

- SET OF IDEAS
- SERIES OF EXAMPLES

FOR CLEAR THINKING IN THE CONSTRUCTION OF PROGRAMS.

INSTRUCTOR NOTES

AN EQUIVALENT TO THIS CAN BE FOUND IN CIRCUIT DESIGN. ANY LOGIC CIRCUIT, NO MATTER HOW COMPLEX CAN BE CONSTRUCTED USING ONLY THE FOLLOWING THREE (3) GATES

- AND
- OR
- NOT

SECOND NAMES

- BOHM AND JACOPINI TOOK THIS ONE STEP FURTHER
- SHOWED THREE (3) SIMPLE CONTROL STRUCTURES WERE CAPABLE OF EXPRESSING ANY PROGRAM REQUIREMENT
  - SEQUENCE
  - ITERATION
  - SELECTION

INSTRUCTOR NOTES

A LA SADT AND MYERS AND CONSTANTINE

TODAY

"STRUCTURED PROGRAMMING" ALSO ENCOMPASSES STRUCTURED ANALYSIS AND STRUCTURED DESIGN.

INSTRUCTOR NOTES

THE CHANGE IN PROGRAMMER'S ATTITUDE REFERS TO TEACHING HIM/HER SOME TECHNIQUES TO PRODUCE CLEARER, CORRECT CODE AND AN APPRECIATION FOR THE RESULTS OF THESE TECHNIQUES.

PRECISION

- STRUCTURED PROGRAMMING ALLOWS A DEGREE OF PRECISION NEVER BEFORE POSSIBLE
- LARGE PROGRAMS SHOULD NOW HAVE MBF OF ONE (1) YEAR OR SO
- THIS REQUIRES CHANGE IN PROGRAMMER'S ATTITUDE

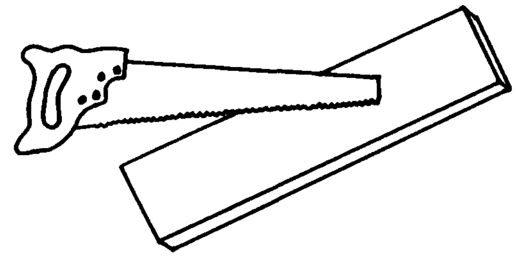
INSTRUCTOR NOTES

THESE ARE TWO (2) KINDS OF PRECISION

PRECISION

TIC TAC TOE

SAWING A BOARD



COMBINATIONAL

REQUIRES RESOLUTION

MILLS, H.D. MATHEMATICAL FOUNDATIONS FOR STRUCTURED PROGRAMMING FSC 72-6112, IBM, FEBRUARY 1972.

INSTRUCTOR NOTES

ONLY DIFFERENCE BETWEEN COMPUTER PROGRAMMING AND TIC TAC TOE IS THE DEGREE OF COMPLEXITY.

THERE'S FINITE AND FINITE!

DEFINITION

COMPUTER PROGRAMMING IS COMBINATIONAL, REQUIRING CORRECT CHOICES OUT OF FINITE SETS OF POSSIBILITIES AT EACH STEP.

INSTRUCTOR NOTES

NOW LEAD UP TO <u>"THE"</u> DEFINITION OF STRUCTURED PROGRAMMING ON THE NEXT FOIL.

DEFINITION

AS A CHILD LEARNS TIC TAC TOE, HE DEVELOPS THEOREMS CONCERNING CORNER SQUARES, CENTER SQUARES, ETC. AND THE "... SELF DISCIPLINE TO BLOCK POTENTIAL DEFEATS BEFORE BUILDING HIS OWN THREATS ..."

MILLS, H.D. MATHEMATICAL FOUNDATIONS OF STRUCTURED PROGRAMMING FSC 72-6112, IBM, FEBRUARY 1972.

INSTRUCTOR NOTES

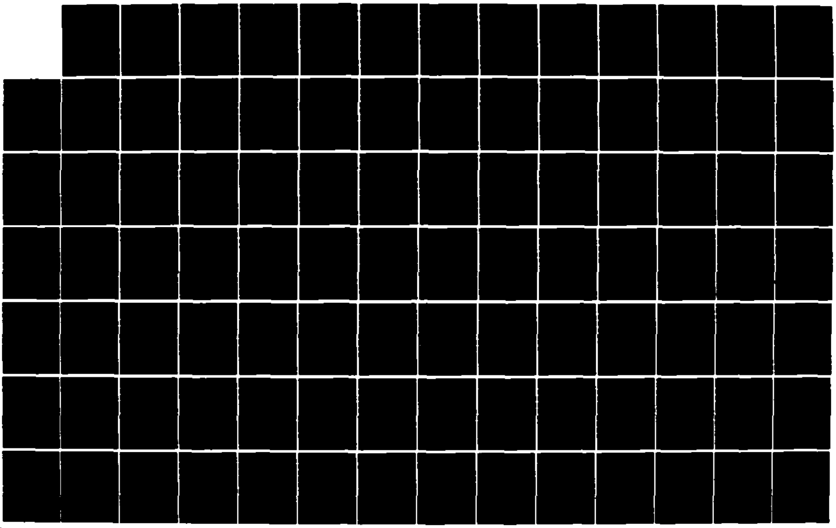
THIS IS ONLY ONE OF A WHOLE HOST OF POSSIBLE DEFINITIONS. BUT THIS DEFINITION IS THE BASIS OF THIS MODULE.

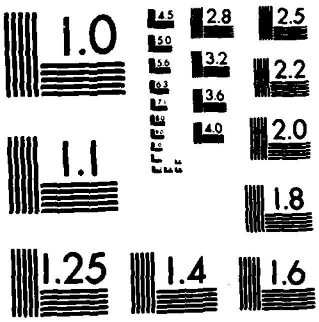
ONE DEFINITION

STRUCTURED PROGRAMMING IS THE <u>THEORY</u> AND <u>DISCIPLINE</u> WHICH PROVIDES A SYSTEMATIC WAY OF DEALING WITH COMPLEXITY IN PROGRAM DESIGN AND DEVELOPMENT WITH A DEGREE OF PRECISION NOT PREVIOUSLY POSSIBLE.

AD-A143 501 ADA (TRADEMARK) TRAINING CURRICULUM PROGRAMMING METHODOLOGY M203 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA JUL 84 DAAB07-83-C-K514 2/8

UNCLASSIFIED F/G 5/9 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

INSTRUCTOR NOTES

IF THE PROGRAMMER DOESN'T KNOW HIS CAPABILITY, HE(SHE)'LL STILL RELY ON THE COMPUTER - HOW TRAGIC!

ONCE HE KNOWS HE KNOWS, HE'LL WRITE CORRECT PROGRAMS.

DEFINITION

- THE PROGRAMMER MUST <u>KNOW</u> HIS/HER CAPABILITY FOR PRECISION PROGRAMMING

- KNOWING THAT YOU KNOW MEANS YOU DON'T HAVE TO GUESS AND HOPE

INSTRUCTOR NOTES

THE FOIL INDICATES THE REFERENCE FOR THE NEXT 3 FOILS.

A SUCCESSFUL APPLICATION

NEW YORK TIMES PROJECT

F.T. BAKER, CHIEF PROGRAMMER TEAM MANAGEMENT OF PRODUCTION PROGRAMMING. IBM SYSTEM JOURNAL NO. 1, 1972.

INSTRUCTOR NOTES

A SURGEON IS ASSISTED BY A STAFF OF SPECIALISTS E.G. ANESTHESIOLOGIST, NURSES, ETC.

THE CHIEF PROGRAMMER DOES THE CRITICAL PART OF THE SYSTEM AND SPECIFIES AND INTEGRATES ALL OTHER PROGRAMMING FOR THE SYSTEM.

NEW YORK TIMES PROJECT

- INFORMATION BANK SYSTEM
- IBM 360/IBM 2321
- CONCEPT ANALOGOUS TO SURGICAL TEAM
  - CHIEF PROGRAMMER
  - STAFF OF SPECIALISTS
    - BACKUP PROGRAMMER
    - LIBRARIAN

INSTRUCTOR NOTES

FUNCTIONAL:	A PROJECT MANAGER TO HANDLE LEGAL AND ADMINISTRATIVE REQUIREMENTS CHIEF PROGRAMMER TO WORRY ABOUT THE TECHNICAL ASPECTS.
PPL:	ISOLATE CLERICAL WORK FROM PROGRAMMING.

NEW YORK TIMES PROJECT

FOUR PROGRAMMING MANAGEMENT TECHNIQUES

- FUNCTIONAL ORGANIZATION
- PROGRAM PRODUCTION LIBRARY
- TOP DOWN PROGRAMMING
- STRUCTURED PROGRAMMING

INSTRUCTOR NOTES

PRODUCTIVITY HERE IS ABOUT 2-4 TIMES BETTER THAN NORMAL FIGURES.

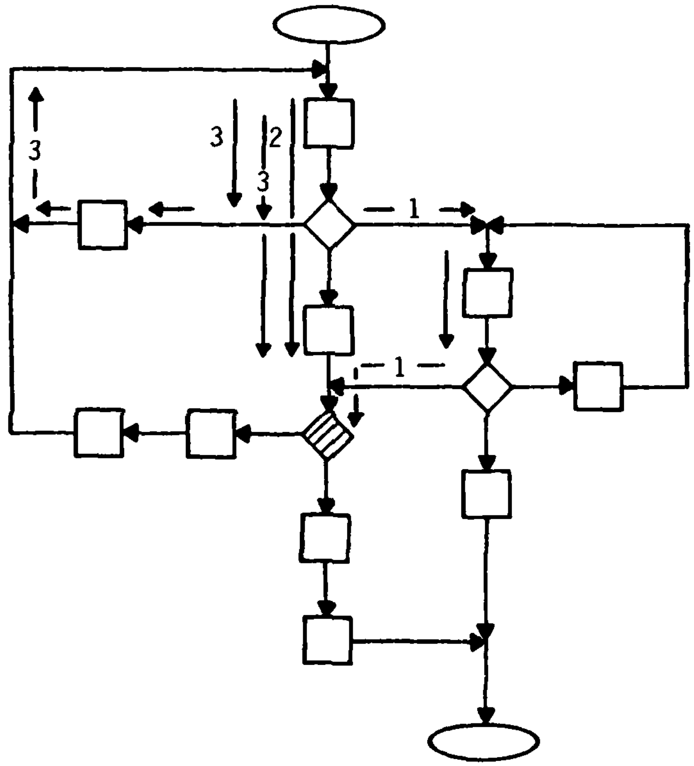
NEW YORK TIMES PROJECT

- APPROXIMATELY 84,000 LINES OF CODE
  - PRODUCTIVITY APPROXIMATELY 35 LINES/DAY

INSTRUCTOR NOTES

POINT OUT THAT THERE IS NO WAY TO DETERMINE WHICH PATH LEADS TO A SPECIFIC BOX, HERE THE SHADED DIAMOND SHAPE.

TYPICAL EARLY PROGRAM



INSTRUCTOR NOTES

BULLET 1 - SEQUENCE, SELECTION, ITERATION

BULLET 4 - DEAD CODE IS UNREACHABLE CODE

A STRUCTURED PROGRAM

- USES A FIXED SET OF STRUCTURES
- EACH STRUCTURE HAS ONE ENTRY AND ONE EXIT
- ANY PROGRAM CAN BE WRITTEN USING THESE THREE (3) STRUCTURES
- NO DEAD CODE
- THE PROGRAM SHALL HAVE ONE ENTRY, ONE EXIT

INSTRUCTOR NOTES

THE STRUCTURE THEOREM STATES THAT ANY PROGRAM CAN BE WRITTEN USING THESE THREE (3) CONTROL LOGICS.

THREE CONSTRUCTS

<u>SEQUENCE</u>

- TWO ACTIONS ARE PERFORMED ONE AFTER ANOTHER

<u>SELECTION</u>

- DECISION IS MADE AND ONE OF SEVERAL ACTIONS IS PERFORMED DEPENDING ON THE RESULT OF THE DECISION

<u>ITERATION</u>

- AN ACTION IS REPEATED A NUMBER OF TIMES. GENERALLY THE ACTION IS TERMINATED WHEN SOME CONDITION IS MET

INSTRUCTOR NOTES

FLOWCHARTS DESCRIBE THE STRUCTURES.

FLOWCHARTS OBEY THE FOLLOWING RULES:

- RULE 1 - ONLY 3 SYMBOLS ALLOWED IN A FLOWCHART
- RULE 2 - FUNCTION SYMBOL HAS 1 ENTRY AND 1 EXIT
- RULE 3 - DECISION SYMBOL HAS 1 ENTRY AND 2 EXITS
- RULE 4 - CONNECTOR SYMBOL HAS 2 ENTRIES AND 1 EXIT

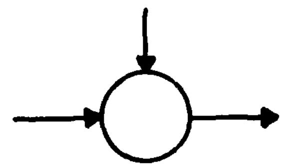
ANY FLOWCHART WHICH OBEYS THESE RULES IS CALLED A STRUCTURED FLOWCHART.

FLOWCHART

RULE 1

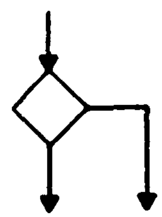
3 SYMBOLS TO CONSTRUCT FLOWCHARTS

CONNECTOR



FUNCTION

DECISION



INSTRUCTOR NOTES

HARLAN MILLS DEVELOPED AN ALGORITHM WHICH WOULD CONVERT THE PREVIOUS SPAGHETTI CODE (VG 2-17) INTO A SIMPLER CONSTRUCTION.

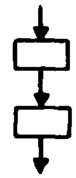
DON'T NEED A goto STATEMENT TO IMPLEMENT. A FORWARD JUMP CAN BE ACHIEVED USING SELECTION, A BACKWARD JUMP CAN BE ACHIEVED WITH A LOOP.

USING THESE CONTROL CONSTRUCTS CREATES MORE READABLE AND UNDERSTANDABLE CODE. ALWAYS KNOW WHICH PATH LED TO A SPECIFIC BOX.

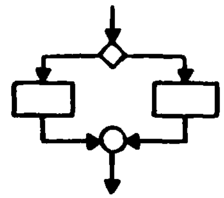
POINT OUT SINGLE INPUT, SINGLE OUTPUT.

THREE CONSTRUCTS

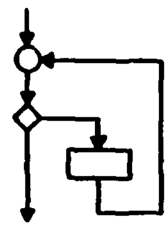
SEQUENCE



SELECTION



ITERATION



INSTRUCTOR NOTES

THREE STATEMENTS EXECUTED IN FIXED ORDER.

VALUE IS MODIFIED BY PROCESS.

BASIC CONSTRUCT

SEQUENCE

```
begin -- Do_Something
   Get (Value);
   Process (Value);
   Put (Value);
end Do_Something;
```

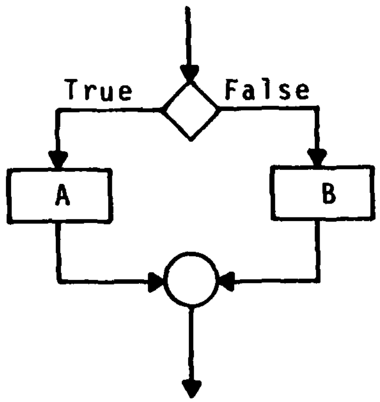
INSTRUCTOR NOTES

IF CONDITION IS TRUE, ONE ACTION IS PERFORMED. IF CONDITION IS FALSE, ANOTHER ACTION IS PERFORMED.

BASIC CONSTRUCT

SELECTION

IF-THEN-ELSE




```
if Condition then
   A;
else
   B;
end if;
```

```
if Condition then
   A;
end if;
```

INSTRUCTOR NOTES

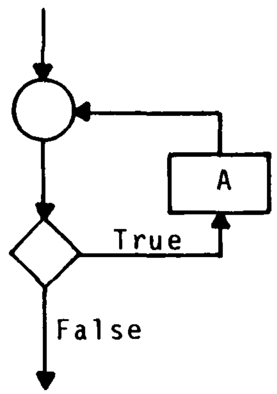
THE TEST IS PERFORMED <u>BEFORE</u> THE ACTION.

IF THE CONDITION IS FALSE WHEN YOU ENTER, THE ACTION WILL NOT BE PERFORMED.

BASIC CONSTRUCT

ITERATION

DO-WHILE




```
while Condition
loop
   A;
end loop;
```

INSTRUCTOR NOTES

EXTENDED STRUCTURES CAN BE MORE CONVENIENT WITHOUT SEVERELY COMPROMISING THE ADVANTAGES OF STRUCTURED PROGRAMMING.

EXTENDED STRUCTURES

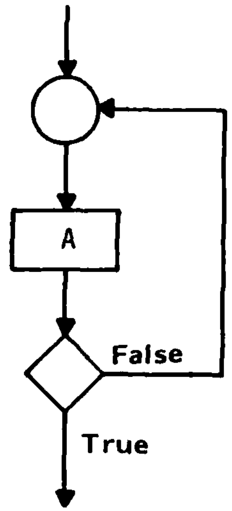
- TWO (2) EXTENDED STRUCTURES
  - DO-UNTIL
  - CASE

INSTRUCTOR NOTES

TEST IS PERFORMED AFTER THE ACTION. THEREFORE IF THE CONDITION IS FALSE WHEN YOU START, THE ACTION IS PERFORMED ONE (1) TIME.

EXTENDED STRUCTURE

DO-UNTIL




```
loop
   A;
   exit when Condition;
end loop;
```

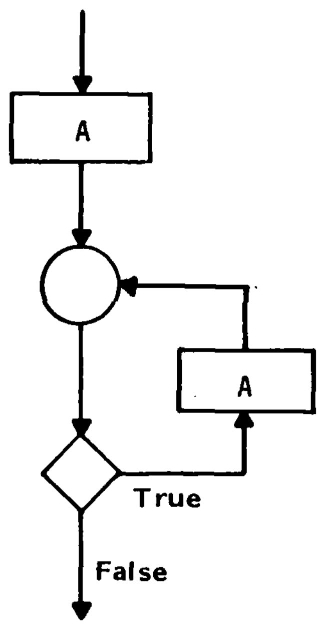
INSTRUCTOR NOTES

THE DO-UNTIL CAN BE EXPRESSED IN TERMS OF THE DO-WHILE.

POINT OUT THE NEGATION OF THE Condition.

EXTENDED STRUCTURES

DO-UNTIL CONSTRUCTED FROM DO-WHILE

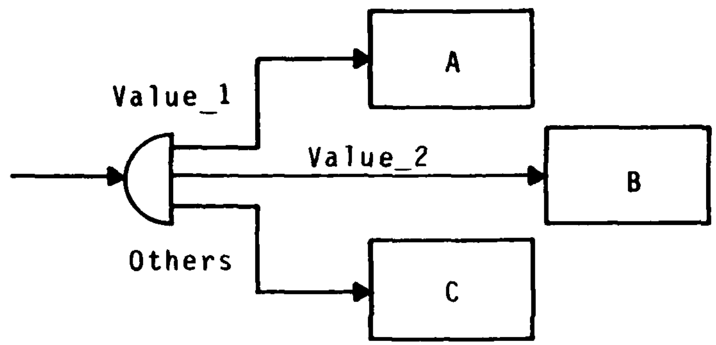



```
A;
while not Condition
Loop
   A;
end loop;
```

AN EXTENSION OF THE IF-THEN-ELSE STRUCTURE.

EXTENDED STRUCTURES

CASE



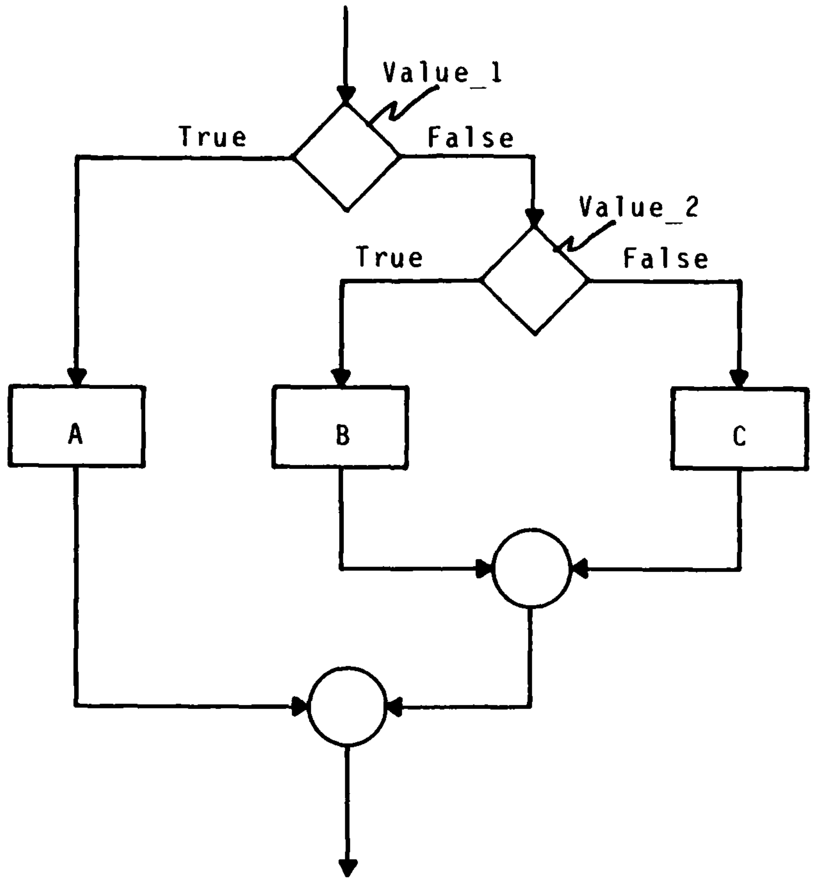

```
case Discrete_Expression is
   when Value_1  =  A;
   when Value_2  =  B;
   when others   =  C;
end case;
```

INSTRUCTOR NOTES

THE CASE CAN BE CONSTRUCTED FROM NESTED IF-THEN-ELSE.

EXTENDED STRUCTURE

CASE CONSTRUCTED FROM IF-THEN-ELSE



INSTRUCTOR NOTES

SOLUTIONS:

1. DO-WHILE. THIS IS AN INTERACTIVE PROCESS. THE END-OF-FILE FLAG SHOULD BE CHECKED BEFORE PROCESSING.

2. IF-THEN-ELSE. THIS IS A SELECTION.

3. CASE. THIS IS SELECTION WITH FOUR POSSIBLE OUTCOMES.

4. DO-UNTIL.

THE POINT IS THAT PROGRAMMERS MUST LEARN TO THINK IN TERMS OF THESE CONSTRUCTS TO DETERMINE OVERALL MODULE STRUCTURE. WORRY ABOUT HOW END OF FILE IS IMPLEMENTED LATER.

EXERCISES

WHICH STRUCTURE WOULD YOU USE TO IMPLEMENT

1. ASSUMING A FILE OF RECORDS IS OPEN, READ THE CONTENTS OF THE FILE UNTIL END-OF-FILE IS REACHED

2. ADDING NUMBER TO TOTAL IF IT IS POSITIVE, DON'T ADD IF IT IS NEGATIVE

3. COMPUTING WEEKLY PAY FOR FOUR DIFFERENT CATEGORIES OF EMPLOYEES

4. FIND THE LARGEST NUMBER, N, SUCH THAT $N! < X$, WHERE $X \geq 0$.

INSTRUCTOR NOTES

GOTO FREE PROGRAMMING

- CONTROL STRUCTURES ELIMINATE NEED FOR GOTOs

INSTRUCTOR NOTES

FORWARD JUMPS ALL CODED USING SELECTION.

FORWARD JUMP

GOTO

```
              if p then
                 goto LABEL_1;
              end if;
              Action_1;
<<LABEL_1>> Action_2;
```

=>

SELECTION

```
if not p then
   Action_1;
end if;
Action_2;
```

INSTRUCTOR NOTES

BACKWARD JUMP IMPLEMENTED USING LOOP.

BACKWARD JUMP

GOTO

```
<<LABEL_1>>if End_of_File then
              goto DONE;
           end if;
           Get (Data);
           Process (Data);
           goto LABEL_1;
<<DONE>>   Next_Action;
```

DO-WHILE

```
while not End_of_File
loop
   Get (Data);
   Process (Data);
end loop;
Next_Action;
```

INSTRUCTOR NOTES

SOMETIMES IN THE COMPLEXITY OF THE SYSTEM ONE FINDS THIS:

ASK THE CLASS TO STRUCTURE THIS. THE SOLUTION IS:

```
Action_1;
Action_2;
Action_3;
Action_4;
Action_5;
```

HOWEVER ...

```
              Action_1;
              goto LABEL_2;
<<LABEL_1>>   Action_3;
              goto LABEL_3;
<<LABEL_2>>   Action_2;
              goto LABEL_1;
<<LABEL_3>>   Action_4;
              Action_5;
```

INSTRUCTOR NOTES

THE KEY POINT IS RELIABILITY.

WHY STRUCTURED PROGRAMMING?

- STRUCTURED PROGRAMMING RESULTS IN SIMPLER CODE
- SIMPLER CODE IS MORE RELIABLE

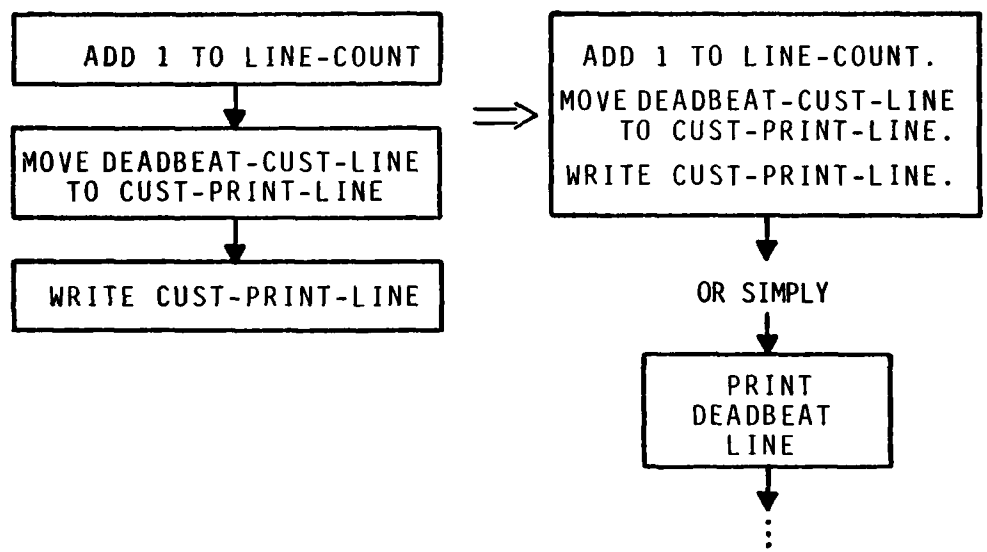
INSTRUCTOR NOTES

ABSTRACTION

- ONE INPUT - ONE OUTPUT SUPPORTS ABSTRACTION

THESE THREE (3) ACTIONS CAN BE SUMMARIZED AS ... 1 ACTION

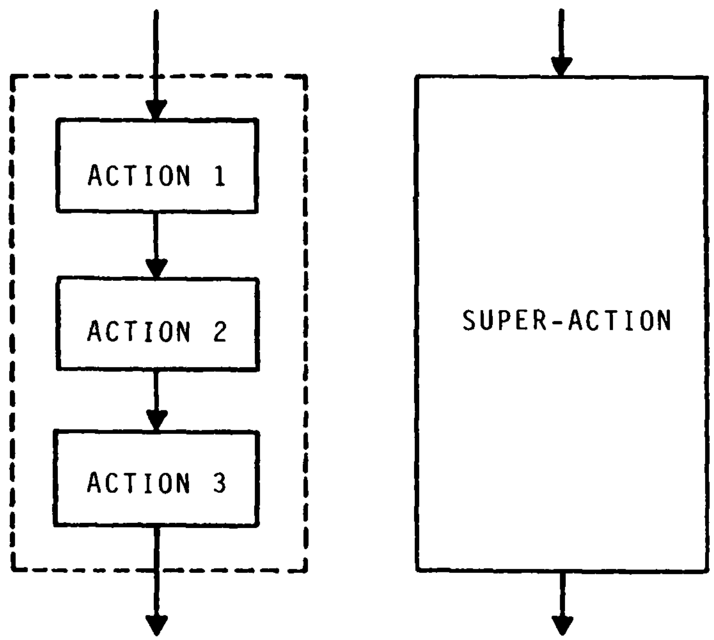
IN OTHER WORDS,



THIS ALLOWS ANY PIECE OF A PROGRAM TO BE TREATED LIKE A SINGLE STATEMENT.

SEQUENCE

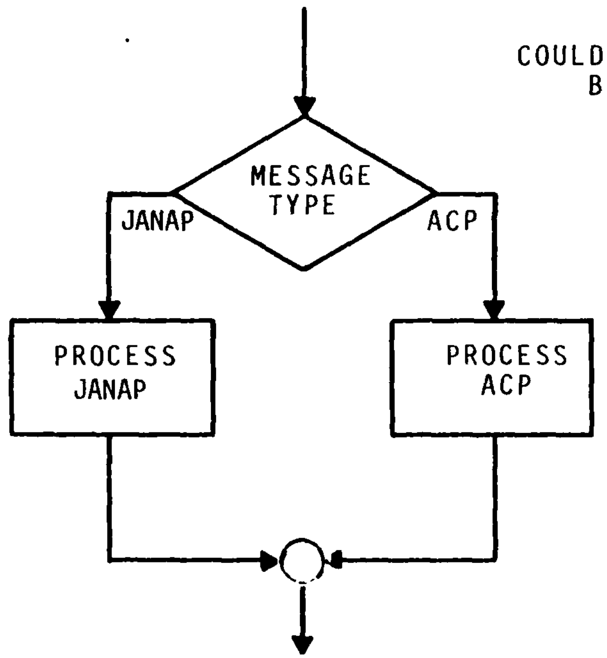
ANY SEQUENCE OF ACTIONS IS ITSELF AN ACTION, E.G.,



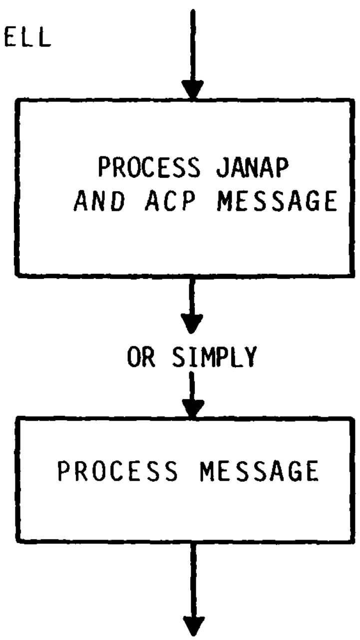
INSTRUCTOR NOTES

THIS CHOICE OF ACTIONS CAN BE SUMMARIZED AS ... 1 ACTION

IN OTHER WORDS,

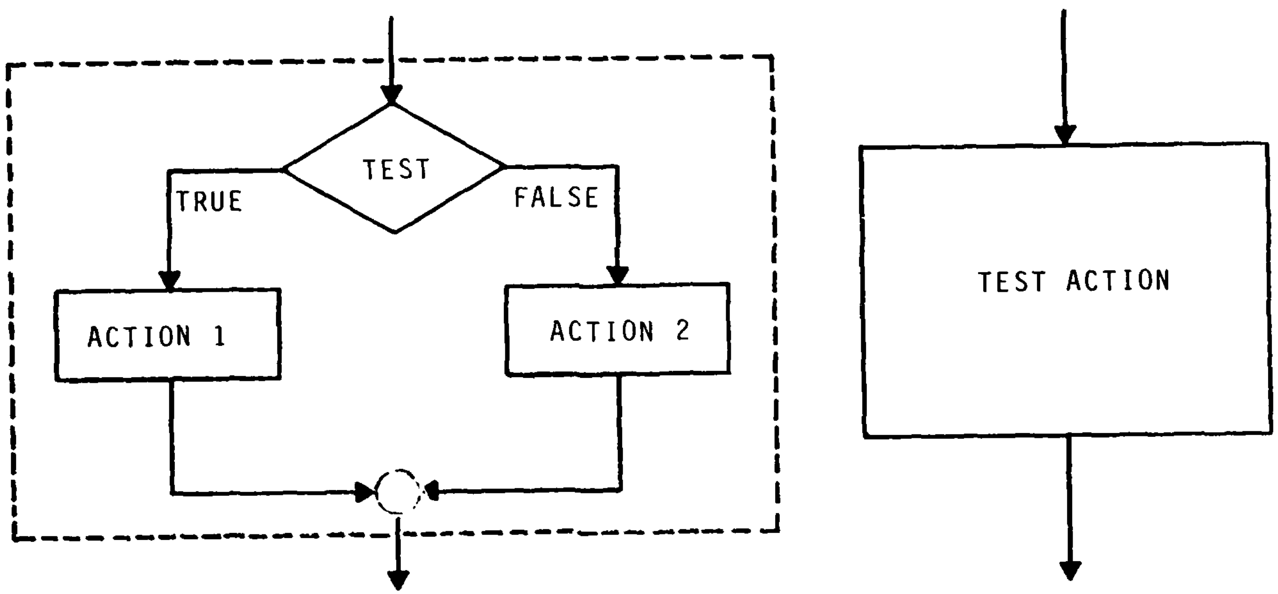


COULD JUST AS WELL BE DRAWN AS



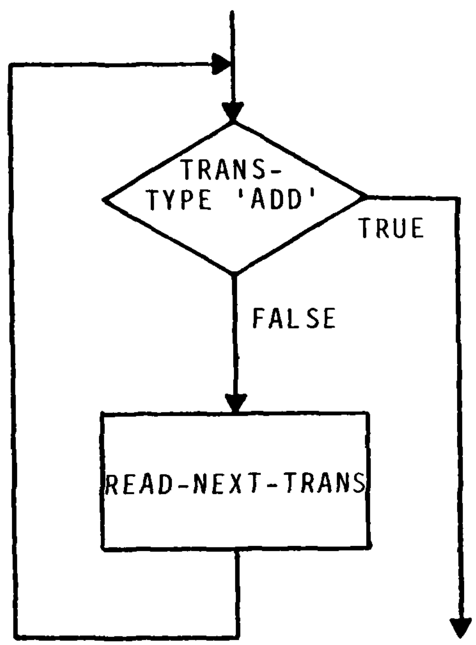
SELECTION

ANY CHOICE OF ACTIONS IS ITSELF AN ACTION, E.G.,

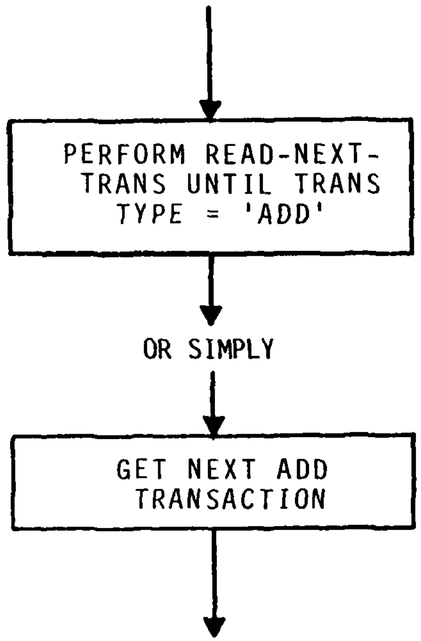


THIS REPETITION OF ACTIONS CAN BE SUMMARIZED AS ... 1 ACTION

IN OTHER WORDS,

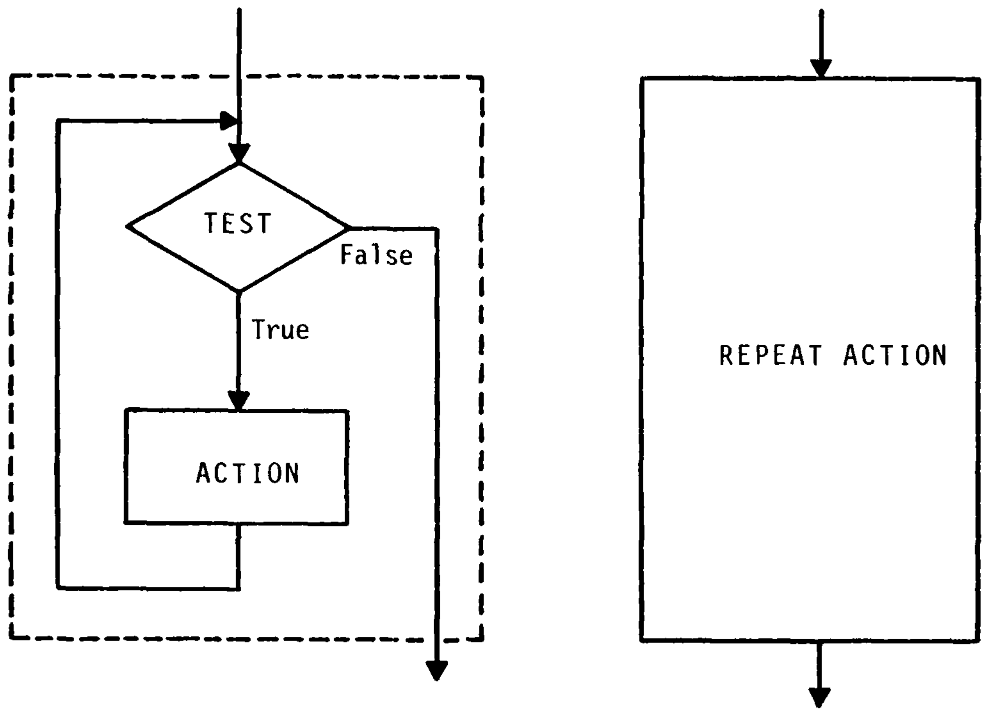


COULD JUST AS WELL
BE DRAWN AS



ITERATION

<u>ANY REPETITION OF ACTIONS IS ITSELF AN ACTION, E.G.,</u>



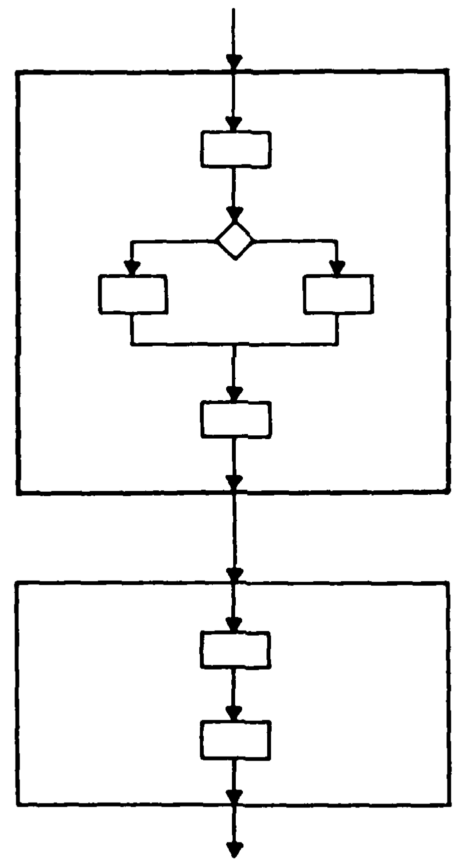
INSTRUCTOR NOTES

THE BIG (AND BOLD) BOX IS JUST ANOTHER NORMAL-SIZED RECTANGLE ON A FLOWCHART AT THE NEXT-HIGHEST LEVEL.

COMPLICATED OPERATIONS CAN BE VIEWED AS A SINGLE OPERATION. DRAW A BOX ANYWHERE AND IT BECOMES VIEWED AS A SINGLE OPERATION WITH ONE INPUT AND ONE OUTPUT.

THE BASIC RULE

AND ANY OF THESE RULES CAN BE NESTED HIERARCHICALLY ...



INSTRUCTOR NOTES

GIVE THE STUDENTS A FEW MINUTES TO LOOK AT THIS CODE. THEN HAVE THEM STRUCTURE IT.

SOLUTION FOLLOWS ON NEXT VIEWGRAPH.

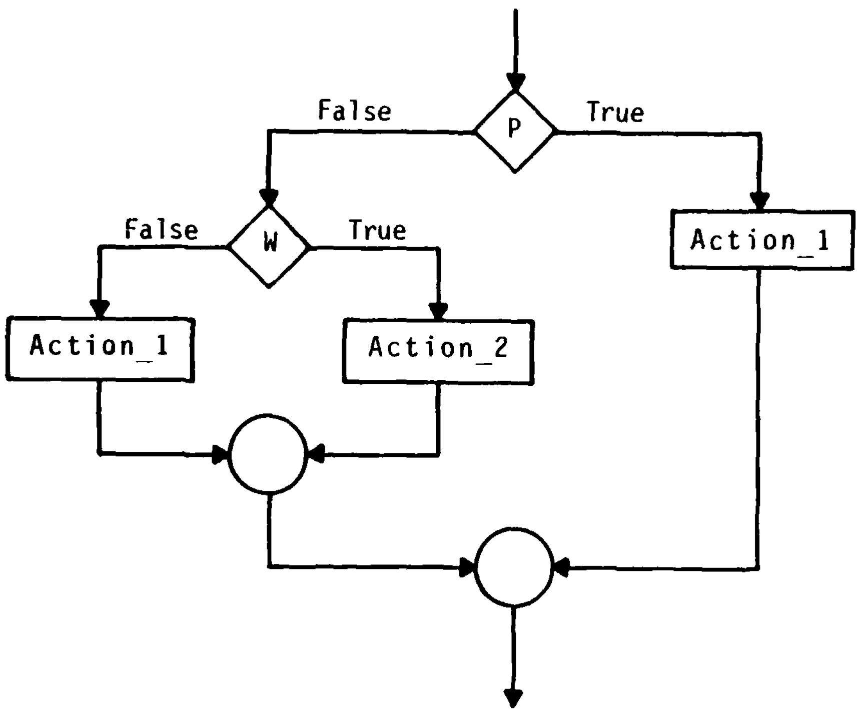
EXERCISE

```
           if p then
              goto LABEL_1;
           elsif w then
              goto LABEL_2;
           end if;
<<LABEL_1>>Action_1;
           goto LABEL_3;
<<LABEL_2>>Action_2;
<<LABEL_3>>...
```

DRAWING A FLOW CHART SHOWS THE STRUCTURE.

FIRST STEP

DRAW A FLOWCHART



INSTRUCTOR NOTES

NOTICE TWO CALLS TO Action_1. CAN THIS BE AVOIDED?

FIRST SOLUTION

```
if p then
   Action_1;
elsif w then
   Action_2;
else
   Action_1;
end if;
```

INSTRUCTOR NOTES

SOMETIMES RETHINKING LOGIC HELPS.

REFER TO THE FLOWCHART TO FIND THE CONDITION UNDER WHICH Action_1 IS PERFORMED. POINT OUT HOW THF GOTO-FREE CODE MAKES IT CLEAR UNDER WHAT CONDITION Action_1 IS PERFORMED, AND SO MAKES THE CODE EASIER TO UNDERSTAND

NOTE THAT THE ORIGINAL CONDITION CAN BE SIMPLIFIED.

SECOND SOLUTION

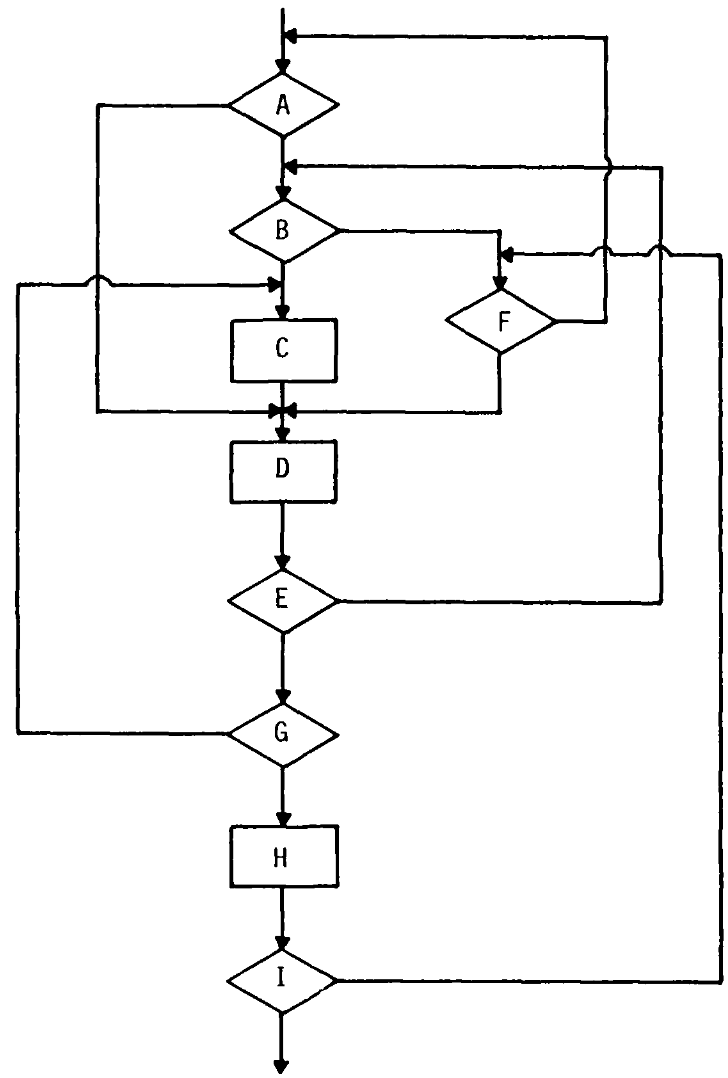
```
if p or (not p and not w) then      -- p or not w
   Action_1;
else
   Action_2;
end if;
```

INSTRUCTOR NOTES

IT'S IMPOSSIBLE TO FOCUS ON ONE SINGLE PART OF THE PROGRAM WITHOUT LOOKING AT THE WHOLE PROGRAM.

A TRADITIONAL FLOWCHART

DIFFICULT TO RELATE PROGRESS OF PROGRAM TO TEXTUAL CODE



INSTRUCTOR NOTES

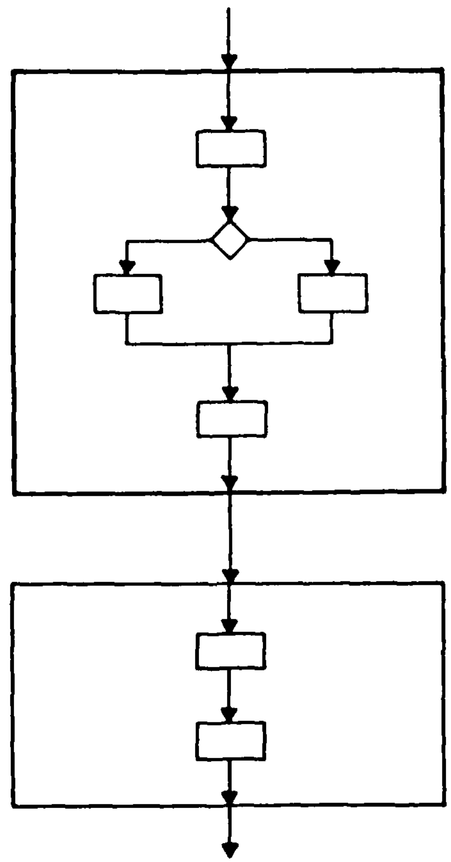
COMPLICATED OPERATIONS CAN BE VIEWED AS A SINGLE OPERATION. DRAW A LARGE BOX ANYWHERE AND IT IS VIEWED AS A SINGLE OPERATION WITH ONE INPUT AND ONE OUTPUT.

ABSTRACTION

BECAUSE THESE RULES CAN BE NESTED HIERARCHICALLY ...

PROGRESS OF PROGRAM

CAN BE MAPPED TO CODE

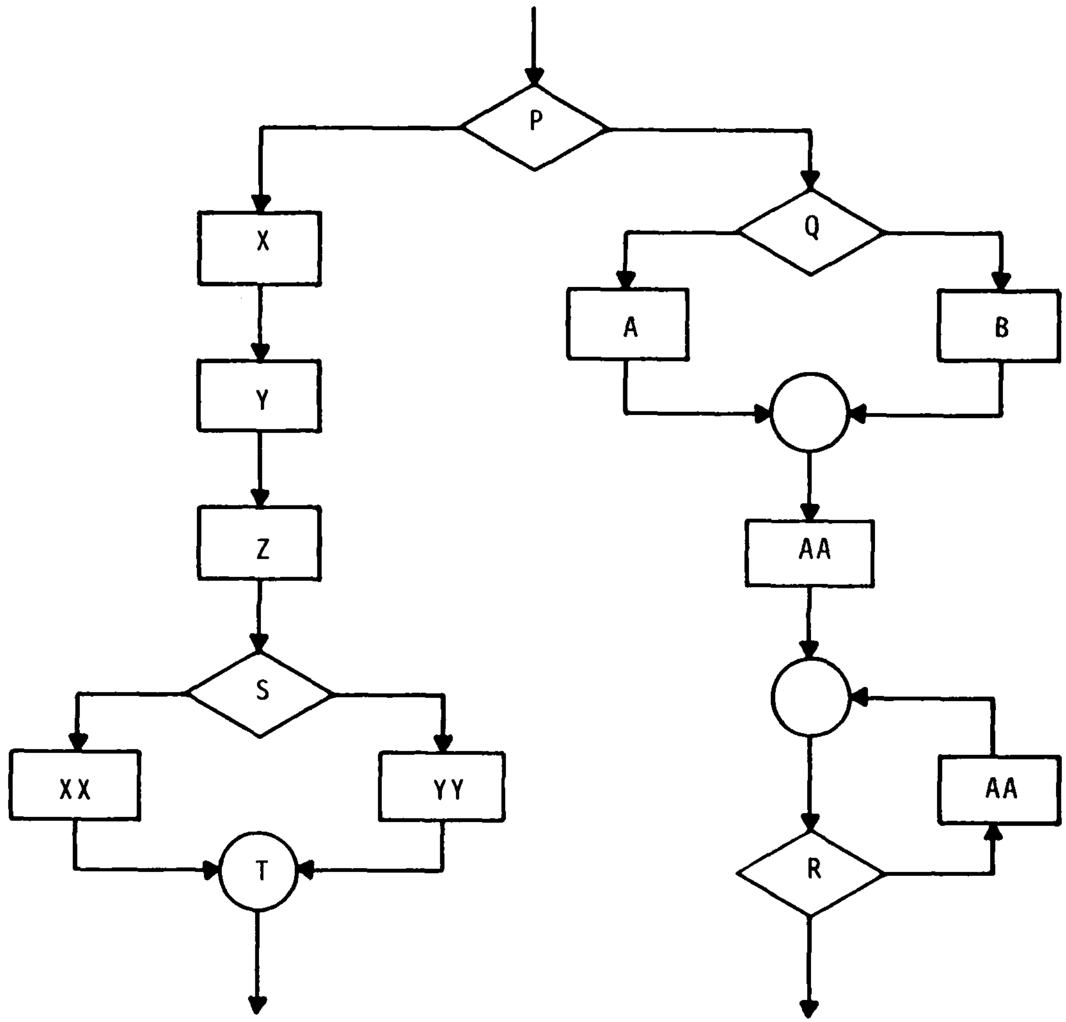


INSTRUCTOR NOTES

HAVE THE STUDENT DRAW BOXES AROUND CONSTRUCTS TO INDICATE ONE OF SEVERAL POSSIBLE NESTINGS. SOLUTION ON NEXT SLIDE.

IN THEORY NESTING CAN GO DOWN TO SEVERAL LEVELS.

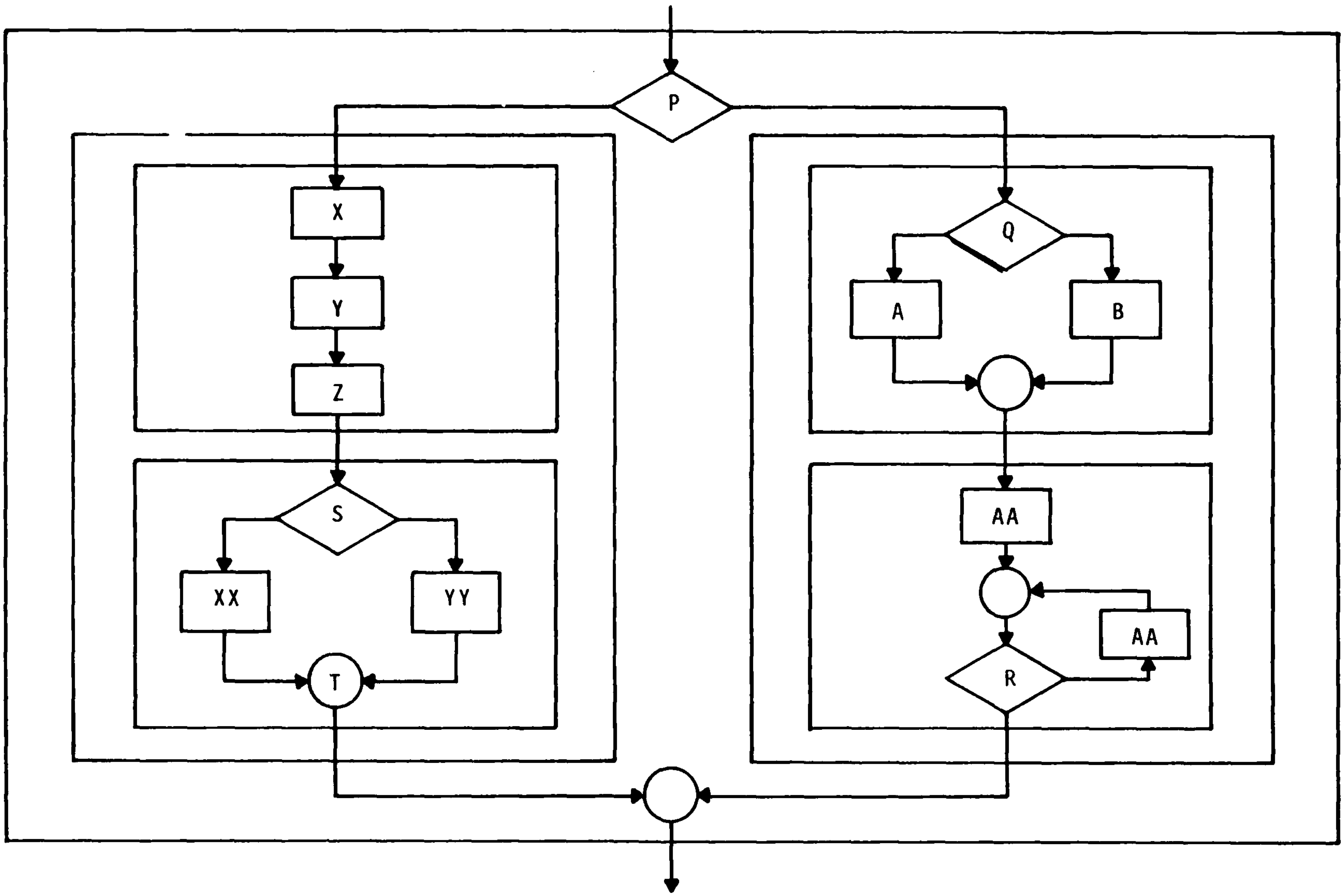
AN EXERCISE



INSTRUCTOR NOTES

THESE ARE THE <u>MAJOR</u> "BLACK BOXES."

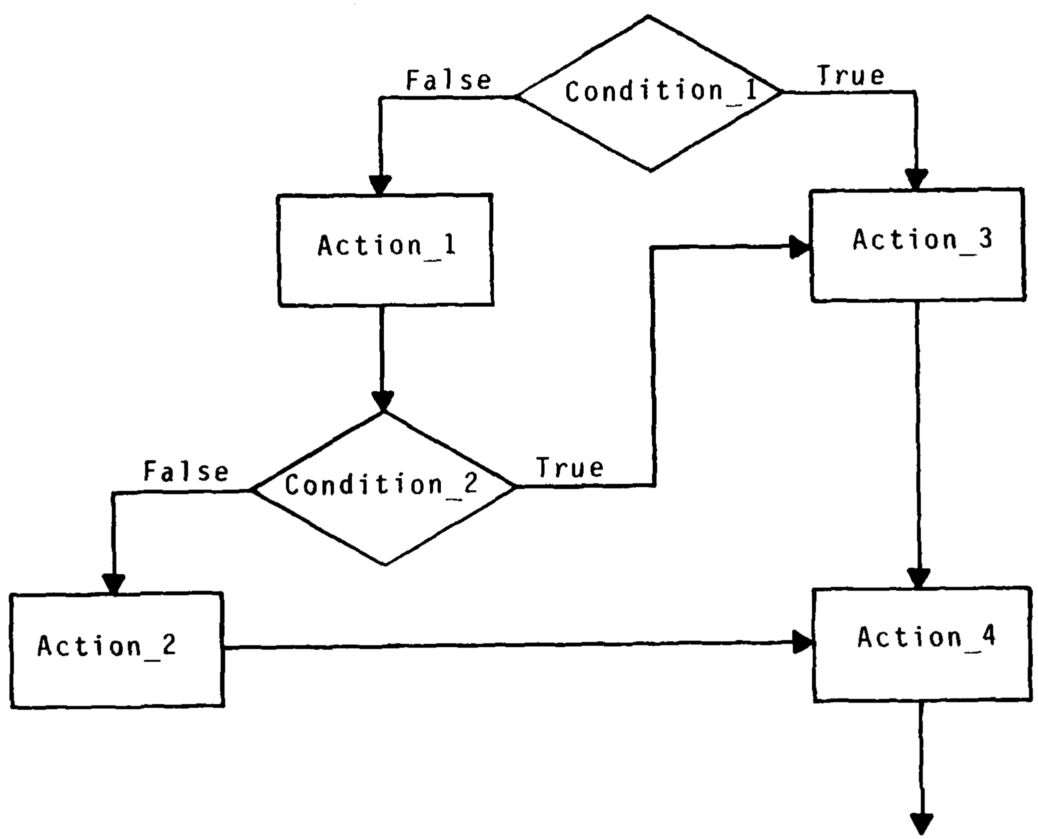
A SOLUTION



VG 817

ASK THE STUDENTS TO REWRITE THE CODE ELIMINATING THE GOTOs.

THE FLOWCHART FOR THE CODE IS:



EXERCISE

```
              if Condition_1 then
                 goto LABEL_1;
              end if;
              Action_1;
              if Condition_2 then
                 goto LABEL_1;
              else
                 Action_2;
                 goto LABEL_2;
              end if;
<<LABEL_1>>   Action_3;
<<LABEL_2>>   Action_4;
```

INSTRUCTOR NOTES

POINT OUT THAT THERE ARE TIMES WHERE IT IS REQUIRED TO DUPLICATE CODE IN ORDER TO ACHIEVE A STRUCTURED PROGRAM.

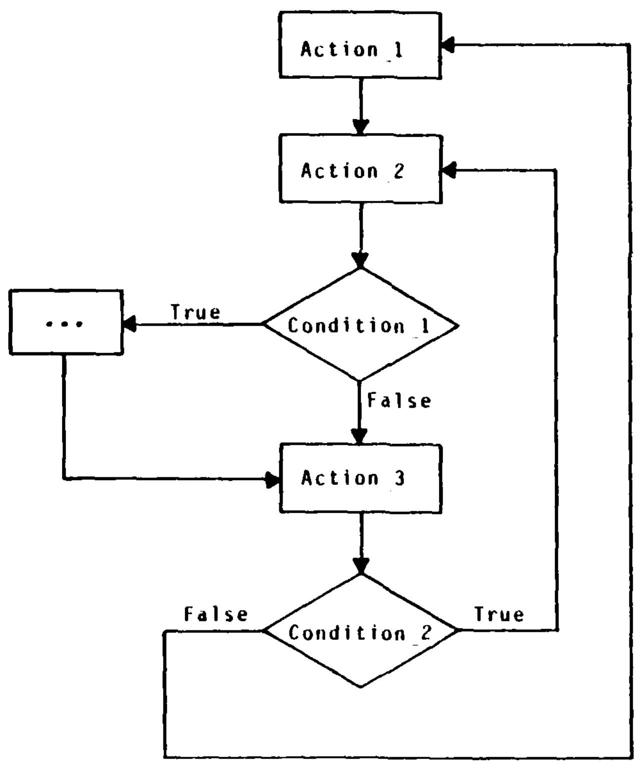
POINT OUT USE OF SELECTION CONSTRUCT TO ELIMINATE FORWARD JUMP.

SOLUTION TO EXERCISE 1

```
if Condition_1 then
   Action_3;
else
   Action_1;
   if Condition_2 then
      Action_3;
   else
      Action_2;
   end if;
end if;
Action_4;
```

ASK THE STUDENT TO REWRITE THE CODE ELIMINATING THE GOTOs.

THE FLOWCHART FOR THIS IS:



EXERCISE 2

```
<<LABEL_1>>  Action_1;
<<LABEL_2>>  Action_2;
             if Condition_1 then
                 --
             end if;
             Action_3;
             if Condition_2 then
                goto LABEL_2;
             else
                goto LABEL_1;
             end if;
```

INSTRUCTOR NOTES

POINT OUT THAT BACKWARD JUMP HAS BEEN IMPLEMENTED WITH A LOOP.

SOLUTION TO EXERCISE 2

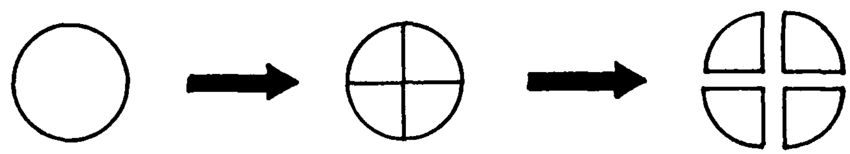
```
Action_1;
loop
   while Condition_2
   loop
      Action_2;
      if Condition_1 then
         --
      end if;
      Action_3;
   end loop;
   Action_1;
end loop;
```

INSTRUCTOR NOTES

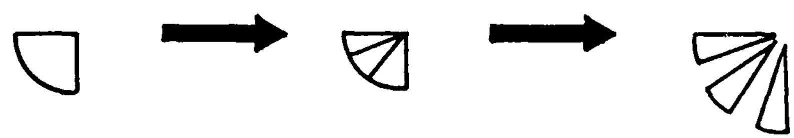
WHAT WE ARE REALLY TALKING ABOUT IS STEPWISE REFINEMENT.

STEP-WISE REFINEMENT

<u>STEP-WISE REFINEMENT</u> IS THE BREAKING OF A PROBLEM INTO SEVERAL SMALLER PIECES:



THEN EACH PIECE CAN BE INVESTIGATED SEPARATELY, AND POTENTIALLY BROKEN INTO STILL SMALLER PIECES:



INSTRUCTOR NOTES

THE LIMITS ARE 7 ± 2.

STEP-WISE REFINEMENT

BUT THE NUMBER OF PIECES IN EACH BREAK UP HAS A LIMIT WITHIN THE BOUNDARIES OF HUMAN COMPLEXITY LIMITS.

ANY METHODOLOGY USING STEP-WISE REFINEMENT LIVES WITHIN HUMAN COMPLEXITY LIMITS.

INSTRUCTOR NOTES

HERE, DIRECTIONS ARE WAY TOO DETAILED TO BEGIN WITH.

STEP-WISE REFINEMENT

A QUESTION ...

Q: HOW DO I GET FROM MY HOME IN ASTORIA, QUEENS, TO 77 OCEAN BOULEVARD, FARGO, NORTH DAKOTA?

A: WELL FIRST YOU GET INTO YOUR CAR, START THE ENGINE, DRIVE DOWN TO THE END OF YOUR STREET, MAKE A LEFT, THEN MAKE THE SECOND RIGHT AND CONTINUE UNTIL YOU COME TO THE THIRD SET OF LIGHTS. THEN YOU ------

Q: AARGHH!

WHAT'S WRONG WITH THIS CONVERSATION?

INSTRUCTOR NOTES

HERE THE DIRECTIONS GO FROM GENERAL TO SPECIFIC; MUCH MORE TOP-DOWN.

NOW CONTRAST <u>THIS</u> CONVERSATION WITH THE FORMER ...

Q: HOW DO I GET FROM MY HOME IN ASTORIA, QUEENS TO 77 OCEAN BOULEVARD, FARGO, NORTH DAKOTA?

A: WELL, THE ROUTE I'D RECOMMEND IS:
INTERSTATE 80 WEST TO ELYRIA, OHIO, THEN
INTERSTATE 90 WEST TO MADISON, WISCONSIN, THEN
INTERSTATE 94 WEST TO FARGO, NORTH DAKOTA.

Q: GOOD, THAT SOUNDS EASY ENOUGH. NOW ALL I NEED IS TO FIND OUT HOW TO GET ONTO INTERSTATE 80.

A: *THE BEST PLACE TO GET ON IS AT THE GW BRIDGE AND* THE BEST WAY FOR YOU TO GET TO THE GW BRIDGE IS VIA THE TRIBOROUGH BRIDGE AND THE HARLEM RIVER DRIVE.

Q: OK, FINE! I KNOW HOW TO DO THAT. BUT WHEN I GET TO FARGO, HOW DO I FIND OCEAN BOULEVARD?

A: WELL, I'M AFRAID I CAN'T HELP YOU THERE. BUT I DO KNOW THAT THERE'S AN INFORMATION CENTER ON 94 JUST BEFORE YOU REACH FARGO. YOU CAN GET ALL THE DETAILS THERE.

Q: THANK YOU VERY MUCH. I'LL SEND YOU A CARD!

THE TOP IS NOT ALWAYS THE BEGINNING, NO METHODOLOGY TELLS YOU HOW TO FIND THE TOP OR THE BEGINNING.

STEP-WISE REFINEMENT

OFTEN, THE HARDEST TASK IN STEP-WISE REFINEMENT IS FINDING THE RIGHT "TOP."

NO METHODOLOGY TELLS YOU WHAT THE "TOP" OF THE SYSTEM IS.

INSTRUCTOR NOTES

THE GENERALIZED PROCEDURE DOES NOT ADDRESS WHEN WRITING THE CARRY IN THE "LAST" COLUMN (FIRST OF THE ANSWER) WHEN THERE ARE NO MORE COLUMNS TO ADD.

$$\begin{array}{r} 592 \\ \underline{636} \\ 1228 \end{array}$$

THIS DIGIT GETS LOST.

STEP-WISE REFINEMENT

HERE'S A SCENARIO FOR ADDING TWO NUMBERS:	AND HERE'S A MORE GENERALIZED PROCEDURE:

$\begin{array}{r} 592 \\ +\underline{236} \\ ??? \end{array}$
 	
1 LOOK AT RIGHTMOST COLUMN
2 ADD 2 TO 6 GIVING 8
3 ENTER 8 IN RIGHTMOST COLUMN OF ANSWER
4 MOVE TO SECOND COLUMN FROM RIGHT
5 ADD 9 TO 3 GIVING 12
6 SEPARATE 12 INTO 2 AND A CARRY OF 1
7 ENTER 2 IN SECOND COLUMN FROM RIGHT TO ANSWER
8 MOVE TO THIRD COLUMN FROM RIGHT
9 ADD 5 TO 2 AND THE CARRY OF 1, GIVING 8
10 ENTER 8 IN THIRD COLUMN FROM RIGHT OF ANSWER
	CLEAR CARRY
	SET CURRENT-COLUMN TO RIGHT-COLUMN
	DO UNTIL NO DIGITS IN CURRENT-COLUMN
	   ADD CARRY AND DIGITS IN COLUMN
	   SPLIT RESULT INTO ANSWER-DIGIT AND CARRY
	   ENTER ANSWER-DIGIT IN CURRENT-COLUMN OF ANSWER
	   MOVE CURRENT-COLUMN LEFT BY 1
	ENDDO

BUT WATCH OUT!! A SINGLE SCENARIO MAY NOT GIVE ENOUGH CLUES!

INSTRUCTOR NOTES

THIS IS A SUMMARY SLIDE.

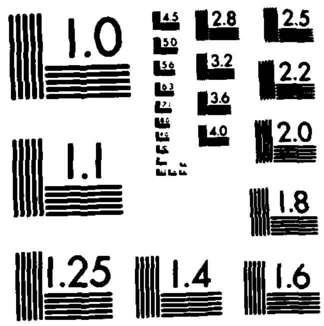
MAIN MESSAGE

SINGLE ENTRY - SINGLE EXIT PERMITS

- ABSTRACTION
- TO CODE MAPPING OF PROGRESS OF A PROGRAM AT RUNTIME
- DETERMINATION OF RELATIONSHIPS OF VARIABLES AT VARIOUS POINTS

AD-A143 501 ADA (TRADEMARK) TRAINING CURRICULUM PROGRAMMING METHODOLOGY M203 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA JUL 84 DAAB07-83-C-K514 3/8

UNCLASSIFIED F/G 5/9 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

INSTRUCTOR NOTES

THIS SUBSECTION DEALS WITH THE THEORETICAL BACK-UP FOR THIS STATEMENT.

STRUCTURING PROGRAMS

IT IS POSSIBLE TO CONVERT AN ARBITRARY PROGRAM TO A STRUCTURED PROGRAM

INSTRUCTOR NOTES

TIME DOES NOT PERMIT GOING THROUGH THE PROOF IN CLASS.

MATHEMATICAL BASIS

- THAT IS NOT AN ARBITRARY STATEMENT BUT IS BACKED UP BY MATHEMATICAL PROOF

- SEE MILLS. MATHEMATICAL FOUNDATIONS FOR STRUCTURED PROGRAMMING FSC 72-6112, IBM, FEBRUARY 1972

INSTRUCTOR NOTES

THIS SUBSECTION BASICALLY COVERS SOME STANDARD PARADIGMS.


BAG
OF
TRICKS

INSTRUCTOR NOTES

BOTH OF THESE WILL BE COVERED IN THIS SUBSECTION.

CODING PARADIGMS

- LOOP PARADIGMS
- CONDITIONAL PARADIGMS

INSTRUCTOR NOTES

WE'RE GOING TO LOOK AT LOOP PARADIGMS.

LOOP PARADIGMS

ADA SUPPLIES

- SIMPLE LOOP
- FOR LOOP
- WHILE LOOP
- EXIT STATEMENTS

WHICH ONE IS APPROPRIATE DEPENDS ON SPECIFIC PROBLEM

INSTRUCTOR NOTES

WHEN A MESSAGE IS BOUND FOR A PARTICULAR DESTINATION, THE ARRAY MAY BE SEARCHED TO FIND A LINE GOING TO THE GIVEN DESTINATION. IF FOUND, THE MESSAGE MAY BE ROUTED. IF NOT FOUND, SOME OTHER ACTION MAY BE TAKEN.

A COMMON PROBLEM

- SEARCH AN ARRAY FOR THE OCCURRENCE OF A SPECIFIC VALUE

- E.G. LOCATE A PHYSICAL LINE GOING TO A PARTICULAR DESTINATION - AN ARRAY WITH AN ENTRY FOR EACH LINE HOLDING THE DESTINATION

INSTRUCTOR NOTES

POINT OUT THAT WE ARE EXAMINING LOOP PARADIGMS, NOT SEARCH ALGORITHMS.

A SIMPLER VIEW

- SEARCH AN ARRAY FOR A GIVEN VALUE
- IF THE VALUE IS FOUND, A Found ROUTINE IS CALLED AND PASSED THE LOCATION IN THE ARRAY
- IF THE VALUE IS NOT FOUND, A Not_Found ROUTINE IS CALLED

INSTRUCTOR NOTES

THE PROBLEMS FOUND IN ONE SOLUTION MOTIVATE THE SUCCEEDING SOLUTION.

THREE APPROACHES

- WHILE LOOP WITHOUT EXIT
- WHILE LOOP WITH EXIT
- FOR LOOP WITH EXIT

INSTRUCTOR NOTES

A FIRST CUi WHICH IS INCORRECT. A PROBLEM EXISTS IN INCREMENTING Index IN THE LAST PASS THROUGH THE LOOP IF THE VALUE IS NOT PRESENT. A(Index) WILL RAISE Constraint_Error BECAUSE Index = Last_Index_in_Array + .

Ada ALLOWS FOR ANY DISCRETE TYPE TO BE THE INDEX SO THIS EXAMPLE IS LIMITING IN ITS SCOPE. THE POINT IS TO CONCENTRATE ON THE LOOP AND THE FACT THAT WE MUST TEST TO DETERMINE HOW WE EXITED THE LOOP.

IF A STUDENT ASKS ABOUT THE SHORT CIRCUIT CONTROL FORM <u>and then</u>, INDICATE THAT THIS IS COVERED LATER.

WHILE LOOP WITHOUT EXIT

```
while Index <= Last_Index_in_Array and A(Index) /= Value
loop
   Index := Index + 1;
end loop;
-- how did we exit?
if Index <= Last_Index
   Found(Index);       -- found value
else
   Not_Found;          -- fell off end of array
end if;
```

1.1

INSTRUCTOR NOTES

- THE FIRST WAY CAN BE DONE IN Ada BY CHANGING THE CONSTRAINTS IMPOSED ON THE INDEX IN ONE OF SEVERAL WAYS.

- WE'LL INVESTIGATE THE 2ND AND 3RD WAYS.

YOU COULD

- INCREASE THE RANGE OF INDEX
- → ADD LOGIC TO AVOID INCREMENTS AFTER LAST VALUE OF RANGE HAS BEEN TESTED
- → ARRANGE LOOP SO THAT IT STOPS BEFORE LAST VALUE

INSTRUCTOR NOTES

THE TEST IN THE LOOP AVOIDS INCREMENTING INDEX DURING THE LAST PASS.

NEEDED AN EXTRA BOOLEAN VARIABLE, Search_Complete, INITIALIZED TO FALSE.

IF THE LOOP GOES COMPLETELY THROUGH TO THE END WITHOUT FINDING VALUE, Search_Complete IS SET TO TRUE.

THE TEST MUST BE MADE EACH PASS THROUGH THE LOOP TO SEE IF IT IS THE LAST PASS.

ADDING ADDITIONAL LOGIC

```
Search_Complete := False;
while A(Index) /= Value and not Search_Complete
loop
   if Index = Last_Index_in_Array then
      Search_Complete := True;
   else
      Index := Index + 1;
   end if;
end loop;
if not Search_Complete then
   Found(Index);
else
   Not_Found;
end if;
```

1.2

INSTRUCTOR NOTES

THE = TEST IN THE LOOP IS ELIMINATED. THIS PUTS THE TEST TO AVOID THE LAST INCREMENT INTO THE while CLAUSE.

IT CAUSES AN EXTRA TEST FOR A (Index) = Value UNLESS THE VALUE IS FOUND IN THE LAST POSITION. THIS IS PERHAPS TRIVIAL FOR A SINGLE ARRAY VALUE, BUT COULD BE EXPENSIVE FOR OTHER COMPONENT TYPES.

ARRANGING THE LOOP SO IT STOPS BEFORE LAST INDEX POSITION

```
while Index < Last_Index_in_Array and A(Index) /= Value
loop
   Index := Index + 1;
end loop;
if A(Index) = Value then
   Found(Index);
else
   Not_Found;
end if;
```

1.3

INSTRUCTOR NOTES

BASICALLY THE LOOP IS EXITED WHEN THE VALUE BEING SEARCHED FOR IS FOUND.

THIS TESTS ALL ARRAY ELEMENTS UP TO BUT NOT INCLUDING THE LAST ONE.

A DUPLICATE TEST A(Index) = Value IS PERFORMED IF THE LOOP IS EXITED VIA THE EXIT STATEMENT.

IF THE LOOP TERMINATED BY THE while CLAUSE, THIS SAME TEST ON THE LAST ARRAY ELEMENT WILL DETERMINE WHETHER Found OR Not_Found IS CALLED.

WHILE LOOP WITH EXIT

```
while Index < Last_Index_in_Array
loop
   exit when A(Index) = Value;
   Index := Index + 1;
end loop;
if A(Index) = Value then
   Found(Index);
else
   Not_Found;
end if;
```

2.1

INSTRUCTOR NOTES

EFFECTIVELY THIS IS A SIMPLE LOOP WITH AN EXIT IN THE MIDDLE.

ASSUME Value_Found IS OF type Boolean.

AN EXTENSION

```
loop
   Value_Found := A(Index) = Value;
exit when Value_Found or Index = Last_Index_in_Array;
   Index := Index + 1;
end loop;
if Value_Found then
   Found(Index);
else
   Not_Found;
end if;
```

2.2

INSTRUCTOR NOTES

AGAIN - A FIRST INCORRECT CUT.

THE Index OUTSIDE OF LOOP IS DIFFERENT FROM Index LOOP PARAMETER.

Index_Range IS NOT AN ERROR. SINCE WE CAN NOT ASSUME Ada KNOWLEDGE ON THE STUDENT'S PART, USE OF THE ATTRIBUTE 'RANGE IS INAPPROPRIATE HERE. JUST TALK "... THE ENTIRE RANGE OF INDICES."

FOR LOOP WITH EXIT

```
for Index in Index_Range
loop
   exit when A(Index) = Value;
end loop;
if A(Index) = Value then
   Found(Index);
else
   Not_Found;
end if;
```

3.1

INSTRUCTOR NOTES

INTRODUCTION OF I NECESSITATES ASSIGNMENT STATEMENT.

ADD LOOP PARAMETER

```
for I in Index_Range
loop
   Index := I;
exit when A(Index) = Value;
end loop;
if A(Index) = Value then
   Found(Index);
else
   Not_Found;
end if;
```

3.2

INSTRUCTOR NOTES

MOVE THE CALL TO Found WITHIN THE LOOP TO ELIMINATE THE NEED FOR THE EXTRA VARIABLE.

THIS DOES REQUIRE SETTING A BOOLEAN VARIABLE TO DECIDE AT THE END OF THE LOOP WHETHER TO CALL Not_Found.

MOVE THE CALL

```
for Index in Index_Range
loop
   if A(Index) = Value then
      Value_Found := True;
      Found(Index);
exit;
   end if;
end loop;
if not Value_Found then
   Not_Found;
end if;
```

3.3

INSTRUCTOR NOTES

TALK ABOUT THE CHART A LITTLE. EMPHASIZE THAT 1.1 AND 3.1 ARE INCORRECT AND ARE THEREFORE USELESS. A PROGRAMMER'S RESPONSIBILITY IS TO PRODUCE CORRECT CODE.

CLARITY = f(SIMPLICITY).

SUMMARY

SOLUTION / CHARACTERISTIC	1.1	1.2	1.3	2.1	2.2	3.1	3.2	3.3
CORRECT	N	Y	Y	Y	Y	N	Y	Y
CLARITY		L	M	M	M		H	M
SE/SE		Y	Y	N	Y		N	N
EXIT AT TOP		N/A	N/A	Y	N/A		Y	N
EXTRA VARIABLES		N	Y	Y	N		N	N
EXTRA COMPARES		Y	N	N	Y		N	Y
EXTRA COMPUTATION		N	Y	Y	Y		N	Y

1.1 AND 3.1 ARE INCORRECT!!

WE'LL LOOK AT THE FIRST BRIEFLY.

WE'LL LOOK AT THE SECOND A LITTLE MORE IN DETAIL.

CONDITIONAL PARADIGMS

- CASE OVER IF
- SHORT CIRCUIT CONTROL FORMS

CONNECTION STATE EVALUATED ONLY ONCE.

CASE OVER IF

```
DEFINITION:  Connection_State = {Idle, Dialing, Ringing, Busy,
                                 Cradled, Release, ...        }

             case Connection_State is
                when Idle => ...
                when Dialing => ...
                   ...
             end case;
```

INSTRUCTOR NOTES

Connection_State EVALUATED MANY TIMES.

CASE OVER IF

```
if Connection_State = Idle then
   ...
elsif Connection_State = Dialing then
   ...
elsif Connection_State = Ringing then
   ...
end if;
```

INSTRUCTOR NOTES

WE'RE GOING TO INVESTIGATE WHEN THE SHORT CIRCUIT CONTROL FORMS ARE APPROPRIATE.

SHORT CIRCUIT CONTROL

AND THEN

OR ELSE

INSTRUCTOR NOTES

THE REAL PURPOSE OF SHORT CIRCUIT CONTROL FORMS IS TO ENSURE THAT A CONDITIONAL EXPRESSION ALWAYS HAS A WELL DEFINED RESULT.

SHORT CIRCUIT CONTROL

MISCONCEPTION : OPTIMIZE EVALUATION OF CONDITIONAL EXPRESSION

CORRECT CONCEPTION: TO ENSURE THAT A CONDITIONAL EXPRESSION ALWAYS HAS A WELL DEFINED RESULT

INSTRUCTOR NOTES

THE LANGUAGE DOES NOT SPECIFY WHICH OPERAND GETS EVALUATED FIRST.

Distance/Time RAISES Numeric_Error IF Time = 0.

CONDITIONAL PARADIGMS

```
if (Time /= 0.0) and (Distance/Time < 5.2) then
   ...
end if;
```

INSTRUCTOR NOTES

Ada OFFERS A BETTER SOLUTION THAN NESTED IF STATEMENTS.

A TYPICAL SOLUTION

```
if Time /= 0.0 then
   if Distance/Time < 5.2 then
      ...
   end if;
end if;
```

INSTRUCTOR NOTES

IF Time = 0 THE LEFT OPERAND IS FALSE. THIS RESULT IS ENOUGH TO DETERMINE THE RESULT OF ENTIRE EXPRESSION. THE RIGHT OPERAND IS NEVER EVALUATED.

SHORT CIRCUIT

```
if (Time /= 0.0) and then (Distance/Time < 5.2) then
   ...
end if;
```

INSTRUCTOR NOTES

IMAGINE SOME ALGORITHM IS EXECUTED DEPENDING ON THE VALUE OF AN OBJECT BEING ACCESSED.

LET Pointer_To POINT TO A SCALAR OBJECT.

CONDITIONAL PARADIGMS

```
if Pointer_To.all /= Value then
   S1;
else
   S2;
end if;
```

WHAT IF Pointer_To DOESN'T POINT TO ANYTHING?

INSTRUCTOR NOTES

THIS USES A NESTED IF TO FIRST TEST FOR A NULL POINTER, THEN FOR THE VALUE OF THE ACCESSED OBJECT.

ONE WAY

```
if Pointer_To /= null then
   if Pointer_To.all /= Value then
      S1;
   else
      S2;
   end if;
end if;
```

INSTRUCTOR NOTES

WALK THROUGH THE CODE. NOTE THAT S1 AND S2 HAVE BEEN INTERCHANGED.

REWRITTEN USING AND THEN

```
if Pointer_To /= null and then Pointer_To.all /= Value then
   S1;
else
   S2;
end if;
```

WALK THROUGH THE CODE.

REWRITTEN USING <u>OR ELSE</u>

```
if (Pointer_To = null) or else (Pointer_To.all = Value) then
   S2;
else
   S1;
end if;
```

DE MORGAN'S LAWS

$$(p \cap q)' \Rightarrow (p' \cup q')$$

$$(p \cup q)' \Rightarrow (p' \cap q')$$

TO NEGATE AN <u>OR</u> OR <u>AND</u>, NEGATE EACH OPERAND AND CHANGE OPERATORS.

REMEMBER

```
if (Pointer_To.all /= null) and then (Pointer_To.all /= Value) then
   S1;
else
   S2;
end if;

if (Pointer_To.all = null) or else (Pointer_To.all = Value) then
   S2;
else
   S1;
end if;
```

because

not (P and Q) is the same as not P <u>or</u> not Q
not (P or Q) is the same as not P <u>and</u> not Q

WRITING STRUCTURED PROGRAMS REQUIRES A DIFFERENT APPROACH. THIS SUBSECTION SUMMARIZES THE PREVIOUS INFORMATION AND DESCRIBES SOME ADDITIONAL ITEMS TO CONSIDER WHEN WRITING STRUCTURED CODE.

SOME THINGS TO THINK ABOUT

- LOOP TERMINATION
- ELIMINATING MULTIPLE EXITS
- EVALUATION OF BOOLEAN EXPRESSIONS

INSTRUCTOR NOTES

THE ADVANTAGE IS THAT THE <u>EXACT</u> CONDITIONS UNDER WHICH THE LOOP WILL BE TERMINATED ARE EXPLICITLY SPECIFIED IN THE <u>while</u> CLAUSE.

LOOP TERMINATION

- CONVENTIONAL PROGRAMMING
  - LOOP CONTROL CLAUSE SPECIFIES THE <u>NOT FOUND</u> CONDITION (E.G. THE END OF THE ARRAY)
  - WHEN <u>FOUND</u> CONDITION IS DETECTED WITHIN THE LOOP, A BRANCH IS DONE TO A POINT OUTSIDE THE LOOP
- STRUCTURED PROGRAMMING
  - CONTROL CLAUSE MUST CONTAIN THE <u>FOUND</u> AS WELL AS THE <u>NOT FOUND</u>

INSTRUCTOR NOTES

ASSUME LOCATION IS DEFINED AS 0 TO Index'Last AND HAS BEEN PREVIOUSLY DECLARED.

POINT OUT THAT THE CONDITION FOR TERMINATING THE LOOP IS FALLING OFF THE END OF THE ARRAY. IT IS THE NOT FOUND SITUATION WITH AN EXIT OUT OF THE LOOP WHEN FOUND.

NON-STRUCTURED

```
Location := 0;
for Index in 1 .. Last_Index_in_Array -- end of array, the
loop                                  -- not Found condition
   if A(Index) = Value then
      Location := Index;
      goto Found_It;                  -- found, so branch outside
   end if;                            -- the loop
end loop;
return 0;
<<Found_It>>
return Location;
```

INSTRUCTOR NOTES

THE TRADEOFFS ARE SUMMARIZED IN THE TABLE WHICH COMPARES SOLUTIONS 1.1 THROUGH 3.2.

STRUCTURED

WE'VE ALREADY SEEN SEVERAL VERSIONS
AND DISCUSSED THE TRADEOFFS.

INSTRUCTOR NOTES

SORTING OUT "Found" AND "Not Found" CONDITIONS AT THE END OF A LOOP IS JUST ONE EXAMPLE OF THE MORE GENERAL PROBLEM OF ELIMINATING MULTIPLE EXITS.

ELIMINATING MULTIPLE EXITS

- CONVENTIONAL PROGRAMMING
  - gotos USED AT APPROPRIATE POINTS
- STRUCTURED PROGRAMMING
  - SET AUXILIARY VARIABLE AT EACH POINT AN EXIT WOULD BE MADE
  - TEST THE VARIABLE BY AN IF OR CASE STATEMENT OUTSIDE THE LOOP TO DETERMINE NEXT ACTION

INSTRUCTOR NOTES

THE NEXT FOILS GIVE EXAMPLES.

EVALUATION OF BOOLEAN EXPRESSIONS

GOAL

TO EXPLICITLY STATE ALL CONDITIONS UNDER WHICH A SET OF ACTIONS ARE EXECUTED.

REMEMBER

TO USE SHORT CIRCUIT CONTROL FORM TO PREVENT EVALUATION OF OPERANDS THAT WOULD RAISE AN EXCEPTION

INSTRUCTOR NOTES

THE NEXT FOIL INCLUDES THE SPECIFICATION FOR PROCEDURE Merge.

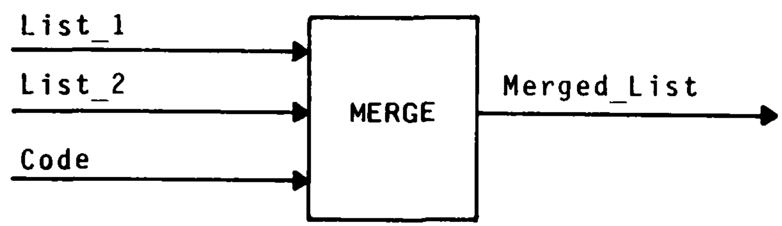
THIS IS AN IN CLASS EXERCISE. HAVE THE STUDENTS COMPLETE THE SOLUTION USING THE CONTROL CONSTUCTS JUST LEARNED. A PDL IS APPROPRIATE HERE, DETAILED CODE IS NOT THE GOAL. THE GOAL IS TO HELP THE STUDENT FOCUS ON WHICH CONSTRUCT TO USE FOR WHICH ACTION.

EXERCISE

MERGE TWO ARRAYS OF INTEGERS. THE ARRAYS ARE OF A FIXED LENGTH BUT CONTAIN AN UNSPECIFIED NUMBER OF VALUES. EACH ARRAY IS ORDERED IN ASCENDING ORDER BUT MAY CONTAIN DUPLICATES. THE LAST ELEMENT OF EACH ARRAY HAS BEEN SET TO A UNIQUE, NON-DATA INTEGER VALUE WHICH WILL BE PASSED TO THE Merge ROUTINE VIA A PARAMETER Code. THE OUTPUT ARRAY SHOULD BE CREATED WITH NO DUPLICATE ELEMENTS AND WITH Code ADDED AT THE END. THE OUTPUT ARRAY IS RETURNED TO THE CALLING ROUTINE AS A PARAMETER.

INSTRUCTOR NOTES

TALK TO THE SPECIFICATION AS A DEFINITION OF A BLACK BOX ACTIVITY.



ASSUME type List_Type is array (Positive range <>) of Integer;

THE SPECIFICATION

```
procedure Merge (List_1, List_2 : in List_Type;
                 Code           : in Integer;
                 Merged_List    : out List_Type;
                 Size           : out Integer);
```

where

```
   List_1, List_2 are the 2 arrays of Integers

   Code           is the integer value of last element

                  in each of the input arrays

   Merged_List    is the merged list also terminated with Code
   Size           is the number of values in the output
                  array not including Code.
```

```
WHILE Any data remains
LOOP
   WHILE data can be taken from array-1
      IF First_entry in output array OR array-1 element 1 = last element_in_output_array
      THEN
         add to output array
         increment size of output array
      END IF;
      increment index
   END LOOP;
   WHILE data can be taken from array-2
   LOOP
      IF First_entry_in output array OR array-2 element/=last element in output array
      THEN
         add to output array
         increment size of output array
      END IF;
      increment index
   END LOOP;
   add code to output array
  Size:  Size-1
END LOOP;
```

NOTES

```
Procedure Merge (etc...) is
   I1, I2:  Integer := 1;
begin -- Merge
   Size := 1
   Merged_List (1) := Code;
   while List_1(I1)/=Code or List_2(I2)/=Code
   loop -- take from array 1 as long as possible
      while List_1(I1)/=Code 2nd (List_2(I2)=Code or List_1(I1)<= List_2(I2)))
      loop
         if Size=1 or List_1(I1)/=Merged_List(Max(Size_1,1)) then
            Merged_List(Size) := List_1(I1);
            Size := Size + 1;
         end if;
         I1 := I1 + 1;
      end loop;
      -- take from array 2 as long as possible
      while (List_2(I2)/=Code and (List_1(I1)=Code or List_2(I2)<= List_1(I1)))
      loop
         if Size = 1 or List_2(I2)/=Merged_List(Max(Size_1,1)) then
            Merged_List(Size) := List_2(I2);
            Size := Size + 1;
         end if;
         I2 := I2 + 1;
      end loop;
   end loop;
   Merged_List(Size) := Code;
   Size := Size-1;
end Merge;
```

NOTES

INSTRUCTOR NOTES

Section 3
CODING STYLE

INSTRUCTOR NOTES

ALLOW 110 MINUTES FOR THE WHOLE SECTION. ALLOCATE THE FOLLOWING TIME TO THE SUBSECTIONS.

- CODING STYLE (30 MINUTES)
- FORMATTING CONVENTIONS (30 MINUTES)
- COMMENTING CONVENTIONS (10 MINUTES)
- NAMING CONVENTIONS (40 MINUTES)

THERE IS A GREAT DEAL OF MATERIAL. TO COVER ALL OF IT THE INSTRUCTOR MUST MOVE BRISKLY THROUGH THE MATERIAL. DO NOT GET BOGGED DOWN IN SYNTAX!

OUTLINE

1. INTRODUCTION
2. STRUCTURED PROGRAMMING
3. **CODING STYLE**
4. ENSURING RELIABILITY
5. REVIEW AND WRAP-UP

THE IMPORTANCE OF GOOD STYLE HAS BEEN UNDERRATED.

GOALS AND NON GOALS OF THIS SECTION

- NOT TO TEACH ADA
  - NOT TO TEACH ALGORITHMS
    - TO TEACH YOU ABOUT STYLE

INSTRUCTOR NOTES

HE WHO THROWS TOGETHER CODE SPENDS A GREAT DEAL MORE TIME DEBUGGING.

AN IMPORTANT POINT

IF IT WAS CODED CLEARLY THE FIRST TIME, THE PROBABILITY OF IT BEING CORRECT IS GREATER AND MAINTENANCE IS EASIER

BUT ...

THIS REQUIRES SELF DISCIPLINE!!!

INSTRUCTOR NOTES

STRESS THAT LARGE EMBEDDED SYSTEMS ARE AROUND FOR A <u>LONG</u> TIME. THEY ARE NOT WRITTEN TO FULFILL A SHORT TERM NEED AND THEN DISCARDED. THEY ARE IN A CONSTANT STATE OF FLUX.

MAINTENANCE IS THE KEY WORD HERE.

ANOTHER IMPORTANT POINT

CLEAN CODE IS EASIER TO MAINTAIN

INSTRUCTOR NOTES

NEXT YEAR YOU WILL BE A

SOMEONE ELSE

GOAL

LEARN TO WRITE AS IF YOU WERE

$\boxed{\text{SOMEONE ELSE}}$

WHAT WE HAVE BEEN TALKING ABOUT LOOSELY IS "STYLE."

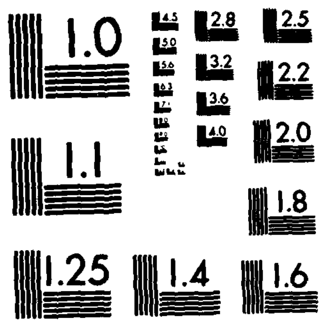
WE CAN ONLY SUGGEST A FEW GUIDELINES IN THE SHORT TIME ALLOTTED.

STYLE

- IS NOT A LIST OF RULES
    - NO "COOKBOOK" APPROACH
- IS AN APPROACH
    - IS AN ATTITUDE
        - THERE ARE GUIDELINES

AD-A143 501 ADA (TRADEMARK) TRAINING CURRICULUM PROGRAMMING METHODOLOGY M203 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA JUL 84 DAAB07-83-C-K514 4/8

UNCLASSIFIED F/G 5/9 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

INSTRUCTOR NOTES

ASK THE CLASS WHAT THE CODE IS DOING. GIVE THE CLASS A FEW MINUTES.

AFTER IT HAS BEEN DETERMINED THAT THIS CODE CREATES AN IDENTITY MATRIX (A MATRIX WITH ONES (1) ON THE DIAGONAL AND ZEROES (0) ELSEWHERE) POINT OUT TO THE CLASS HOW LONG IT TOOK THEM TO DETERMINE THE RESULT OF THE CODE. ASK WHY? WAS IT READABLE?

POINT OUT THAT

1. TOO MUCH TIME SPENT FIGURING OUT WHAT (I/J)*(J/I) IS DOING

2. THE EXPRESSION (I/J)*(J/I) DISTRACTS FROM WHAT IS TO BE DONE BY THE CODE

AN ISSUE - CLARITY

```
for I in 1 .. N
loop
   for J in 1 .. N
   loop
      X(I, J) := (I/J)*(J/I);
   end loop;
end loop;
```

INSTRUCTOR NOTES

ASK THE CLASS WHAT THIS CODE DOES.

POINT OUT THAT THIS VERSION IS

1. MORE READABLE AND UNDERSTANDABLE

2. ISSUE OF IDENTITY MATRIX IS NOT CLOUDED BY "CLEVER" EXPRESSION

```
for Row in 1 .. N
loop
   for Column in 1 .. N
   loop
      Matrix (Row, Column) := 0;
   end loop;
   Matrix (Row, Row) := 1;
end loop;
```

INSTRUCTOR NOTES

BULLET 1

OBSCURE CODE DOESN'T HELP ANYONE.

BULLET 3

POINT OUT THAT IT IS MORE IMPORTANT TO MAKE THE PURPOSE OF THE CODE UNMISTAKABLE THAN IT IS TO DISPLAY TECHNICAL DEPTH BY USING LESS COMMON FEATURES OF A LANGUAGE.

DURING THE DEBUGGING PROCESS WOULD YOU REALLY GO THROUGH ALL THIS OR SKIP OVER IT SAYING "IT LOOKS OK."

MORAL

WRITE CLEARLY - DO NOT BE CLEVER

- OBSCURE CODE MAY NOT DO WHAT YOU THINK IT DOES
- STORAGE AND EXECUTION TIME PROBABLY NOT ALL THAT IMPORTANT ALL THE TIME
- CLARITY MORE IMPORTANT THAN "VIRTUOSITY"

INSTRUCTOR NOTES

DISCUSS WHAT THE FUNCTION DOES.

POINT OUT LOCAL DATA.

ANOTHER ISSUE - TEMPORARY VARIABLES

VERSION 1

```
function Area (Length, Width : Integer) return Integer is
   The_Area : Integer;
begin -- Area
   The_Area := Length*Width;
   return The_Area;
end Area;
```

INSTRUCTOR NOTES

POINT OUT LACK OF LOCAL DATA.

ASK CLASS WHICH IS MORE READABLE?

VERSION 2

```
function Area (Length, Width : Integer) return Integer is
begin -- Area
   return Length*Width;
end Area;
```

INSTRUCTOR NOTES

- THE LESS CHANCE THAT ONE WILL NOT BE PROPERLY INITIALIZED
- THE LESS CHANCE ONE WILL BE UNEXPECTEDLY MODIFIED BEFORE ITS USE
- THE EASIER IT IS TO UNDERSTAND A PROGRAM

MORAL

$\boxed{\text{AVOID TEMPORARY VARIABLES}}$

- INITIALIZATION
- MODIFICATION
- UNDERSTANDABILITY

INSTRUCTOR NOTES

TWO (2) LABELS, THIRTEEN (13) LINES OF CODE, TWO (2) GOTO STATEMENTS ARE USED. ASK THE STUDENTS WHAT THEIR INITIAL IMPRESSION IS?

SURELY <u>SOMETHING</u> IMPORTANT MUST BE HAPPENING!

AFTER MUCH PENCIL PUSHING WE SEE LARGEST IS SET TO THE LARGEST OF X, Y, OR Z.

ANOTHER ISSUE - SIMPLICITY

VERSION 1

```
if X > Y
then
   Largest := X;
   goto THERE;
end if;
Largest := Y;
<<THERE>>
if Largest > Z
then
   goto OUTPUT;
end if;
Largest := Z;
<<OUTPUT>>
   ...
```

INSTRUCTOR NOTES

- LABELS NOT REQUIRED
- GOTOS NOT REQUIRED
- JUST GETTING AT THE MAXIMUM

VERSION 2

```
Largest := X;
if Y > Largest then
   Largest := Y;
end if;
if Z > Largest then
   Largest := Z;
end if;
```

INSTRUCTOR NOTES

REMEMBER THAT UNDERSTANDABILITY IS THE KEY TO RELIABILITY. SIMPLICITY IS A KEY ELEMENT IN ACHIEVING UNDERSTANDABILITY.

MORAL

SAY WHAT YOU MEAN, SIMPLY AND DIRECTLY

INSTRUCTOR NOTES

IF NOT INTERESTED IN EMPHASIZING USE OF RELATIONAL OPERATORS; WRITE A FUNCTION TO RETURN THE MAXIMUM SO THAT THE EXECUTABLE CODE IS MORE READABLE.

POINT OUT THAT READABILITY WAS THE SECOND KEY ELEMENT NEEDED TO ACHIEVE UNDERSTANDABILITY.

VERSION 3

```
Largest := Maximum_of (X, Y, Z);
```

INSTRUCTOR NOTES

DOES "EFFICIENT" MEAN EFFICIENCY IN TERMS OF STORAGE, TIME, OR MACHINE CODE INSTRUCTIONS.

MANY COMPILERS WILL GENERATE THE SAME CODE FOR BOTH EXPRESSIONS.

ANOTHER ISSUE - EFFICIENCY

<u>VERSION 1</u>

```
Temp_1 := X(1) - X(2)*X(2);
Temp_2 := 1.0 - X(2);
Answer := Temp_1*Temp_1 + Temp_2*Temp_2;
--  IT IS MORE EFFICIENT TO COMPUTE
--  Temp_1*Temp_1 THAN TO COMPUTE
--  Temp_1**2.
```

A QUESTION: WHAT DOES "EFFICIENT" MEAN?

INSTRUCTOR NOTES

SOME COMPILERS MIGHT GENERATE FASTER CODE FOR THIS STATEMENT.

DELETION OF TEMPORARY VARIABLES MAKES IT MORE READABLE. AGAIN STATE THE MAXIM OF AVOIDING TEMPORARY VARIABLES WHEN POSSIBLE.

VERSION 2

```
Answer := (X(1) - X(2)**2) **2 + (1.0 - X(2))**2;
```

- IN SOME CASES MORE "EFFICIENT"
- MORE READABLE

INSTRUCTOR NOTES

DO NOT TRY AND OUTSMART THE COMPILER.

EVEN IF VERSION 1 <u>WERE</u> MORE EFFICIENT THERE STILL IS NO REASON TO WRITE SUCH OBSCURE CODE FOR THE EXPRESSION.

PROGRAMS ARE READ MORE OFTEN THAN THEY ARE WRITTEN. IF PEOPLE CANNOT "GRASP" THE INTENT OF THE CODE, THE LONGER IT WILL BE BEFORE THE CODE IS OPERATIONAL.

MORAL

WRITE CLEARLY - DO NOT SACRIFICE
CLARITY FOR EFFICIENCY

INSTRUCTOR NOTES

IF COMPILERS DON'T DO THIS AT COMPILE TIME THEN THERE ARE PROBABLY WORSE INEFFICIENCIES TO BE CONCERNED ABOUT.

IT'S A SHAME TO HAVE TO GO BACK TO THINKING IN BINARY BECAUSE OF ILL-FORMED NOTIONS OF EFFICIENCY.

ANOTHER ISSUE - JOB DESCRIPTION

```
--  NOTE THAT 110010 IN BINARY IS 50
--  IN DECIMAL.
if Number > 2#101111# then
   Put ("~~~");
end if;
```

- COMPUTERS DO CONVERT DECIMAL TO BINARY!

- MOST NOW DO IT AT COMPILE TIME

INSTRUCTOR NOTES

DON'T GET BOGGED DOWN HERE. THERE ARE APPROPRIATE PLACES FOR LITERALS OF BASES OTHER THAN 10 (SPECIFICALLY WHEN WRITING A MACHINE REPRESENTATION SPECIFICATION FOR A TASK ENTRY).

MORAL

LET THE MACHINE DO THE DIRTY WORK

INSTRUCTOR NOTES

ASK THE CLASS TO FIND THE TWO (2) PATTERNS.

- `Sqrt (X1-X2)**2 + (Y1-Y2)**2);`

- `Atan ((4.0*Area)/(Side_A**2 + Side_B**2 - Side_C**2));`

THE ASSIGNMENT STATEMENT FOR AREA HAS THE CLOSING RIGHT PARENTHESIS OMITTED ON PURPOSE. IF THE CLASS DOESN'T SPOT IT, DON'T POINT IT OUT HERE. THE NEXT FOIL WILL ADDRESS THIS POINT.

ANOTHER ISSUE - DUPLICATE CODE

VERSION 1

```
--  Compute Lengths of sides
Side_AB := Sqrt ((X(2) - X(1))**2 + (Y(2) - Y(1)**2));
Side_AC := Sqrt ((X(3) - X(1))**2 + (Y(3) - Y(1)**2));
Side_BC := Sqrt ((X(3) - X(2))**2 + (Y(3) - Y(2)**2));
--  Compute Area
S := (Side_AB + Side_BC + Side_AC)/2.0;
Area := Sqrt (S*(S-SIde_BC)*(S-Side_AC)*(S-Side_AB);
   ...
-- Compute Angles
Alpha := Atan ((4.0*Area)/(Side_AC**2 + Side_AB**2 - Side_BC**2));
Beta  := Atan ((4.0*Area)/(Side_AB**2 + Side_BC**2 - Side_AC**2));
Gamma := Atan ((4.0*Area)/(Side_AC**2 + Side_BC**2 - Side_AB**2));
```

INSTRUCTOR NOTES

THIS IS EASIER TO WRITE.

THIS IS EASIER TO CHANGE.

THIS IS MORE LIKELY TO PRODUCE CORRECT CODE. THE PREVIOUS FOIL HAS AN ERROR IN IT. THE ASSIGNMENT STATEMENT FOR AREA IS MISSING ONE CLOSING PARENTHESIS. IT IS DIFFICULT TO SPOT THE ERROR IN ALL THE CODE.

VERSION 2

DEFINE TWO FUNCTIONS

```
function Side_of (X_1, X_2, Y_1, Y_2 : Integer) return Float;

function Angle_of (Area, Side_A, Side_B, Side_C : Float) return Float;
```

SO THAT YOU CAN WRITE

```
Side_AB := Side_of (X(1), Y(1), X(2), Y(2));
Side_AC := Side_of (X(1), Y(1), X(3), Y(3));
Side_BC := Side_of (X(2), Y(2), X(3), Y(3));
   ...
S := (Side_AB + Side_BC + Side_AC)/2.0;
Area := Sqrt (S*(S-Side_BC)*(S-Side_AC)*(S-Side_AB);
   ...
Alpha := Angle_of (Area, Side_AC, Side_AB, Side_BC);
Beta  := Angle_of (Area, Side_AB, Side_BC, Side_AC);
Gamma := Angle_of (Area, Side_AC, Side_BC, Side_AB);
```

INSTRUCTOR NOTES

FOLLOWING THIS PARADIGM MAKES THE CODE EASIER TO READ AND MODIFY.

ANY OVERHEAD ADDED BY ADDING EXTRA MODULES IS MORE THAN COMPENSATED FOR BY THE EASE OF LATER COMPREHENSION.

MORAL

REPLACE REPETITIVE EXPRESSIONS BY CALLS TO
A COMMON SUBPROGRAM

REVISIT THE EXAMPLE AND DISCUSS THE USE OF PARENTHESES TO AVOID POTENTIAL CONFUSION.

DISCUSS:

1. A*B/2.0*C VERSUS (A*B)/(2.0*C)

2. TERM*(-X**2)/DENOM

   IS IT $(-X)^2$ OR $-(X^2)$

IN TERMS OF HOW THEY INVITE MISUNDERSTANDING.

REVISITED

PARENTHESIZE TO AVOID AMBIGUITY AND ERROR

INSTRUCTOR NOTES

POINT OUT THE DUPLICATE CODE. <u>DON'T GET BOGGED DOWN IN EXPLAINING THE CODE</u>. THE POINT IS THAT THERE IS DUPLICATE CODE, MAKING THE CODE LOOK MORE COMPLEX THAN IT REALLY IS.

THIS IS AN IMPORTANT PARADIGM IN ADA.

THE SAME ISSUE - DUPLICATE CODE

VERSION 1

```
begin -- Some_Procedure
   Put_Line ("Enter first value");
   loop
      begin
         Get (Value_1);
         exit;
      exception
         when Data_Error => Put_Line ("Improper Entry");
                            Put_Line ("Enter only Integer value")
      end;
   end loop;
   Put_Line ("Enter second value");
   loop
      begin
         Get (Value_2);
         exit;
      exception
         when Data_Error => Put_Line ("Improper Entry");
                            Put_Line ("Enter only Integer value");
      end;
   end loop;
   ...
end Some_Procedure;
```

INSTRUCTOR NOTES

DON'T GET BOGGED DOWN IN SYNTAX. THE POINT IS THAT WE HAVE WRITTEN A PROCEDURE AND NOW MAY REPLACE THE EXECUTABLE CODE BY PROCEDURE CALL.

THE MAIN PROGRAM IS CODED ON THE NEXT FOIL.

VERSION 2

```
with Text_IO; use Text_IO;
procedure This_Get (Value : out Integer) is
   --  Proper Instantiation here
begin -- This_Get
   loop
      begin
         Get (Value);
         exit;
      exception
         when Data_Error => Put_Line ("...");
                            Put_Line ("...");
      end;
   end loop;
end This_Get;
```

INSTRUCTOR NOTES

POINT OUT HOW READABLE THE EXECUTABLE CODE IS.

VERSION 2 (CONTINUED)

```
with This_Get;
with Text_IO; use Text_IO;
procedure Do_Something is
   --  necessary declarations
begin -- Do_Something
   Put_Line ("Enter first value");
   This_Get (Value_1);
   Put_Line ("Enter second value");
   This_Get (Value_2);
      ...
end Do_Something;
```

INSTRUCTOR NOTES

FOLLOWING THIS PARADIGM ALSO MAKES THE CODE EASIER TO READ AND MODIFY.

MORAL

REPLACE REPETITIVE STATEMENTS BY CALLS
TO A COMMON SUBPROGRAM

INSTRUCTOR NOTES

TIME DOES NOT PERMIT A FULL TREATMENT. INDICATE TO THE STUDENT THAT IT IS TO HIS/HER BENEFIT TO READ KERNIGHAN AND PLAUGER.

STYLE WRAP-UP

THIS SUBSECTION WAS ADAPTED FROM

ELEMENTS OF PROGRAMMING STYLE.
B.W. KERNIGHAN AND P.J. PLAUGER
MCGRAW HILL

AND ONLY TOUCHES THE SURFACE

INSTRUCTOR NOTES

THIS SUBSECTION DEALS WITH FORMATTING CONVENTIONS.

FORMAT

INSTRUCTOR NOTES

ALTHOUGH THE LANGUAGE REFERENCE MANUAL EXHIBITS SOME FORMATTING CONVENTIONS, CONVENTIONS ARE MOST LIKELY SHOP DEPENDENT.

STATE THAT THESE ARE <u>RECOMMENDED</u>, NOT MANDATED.

FORMATTING CONVENTIONS

- FORMATTING CONVENTIONS CAN IMPROVE READABILITY
- NEED CONVENTIONS

INSTRUCTOR NOTES

<u>PLUS</u>

+ EASIER TO READ
+ READS MORE LIKE ENGLISH

<u>MINUS</u>

- HARD TO FIND RESERVED WORD BURIED IN THE CODE

RESERVED WORDS

VERSION 1

```
BEGIN
   IF Wing < 0.6*Length THEN
      Factor := (1.0 + 0.037);
   ELSE
      Factor := (1.0 + 0.048);
      END IF;
END;
```

VERSION 2

```
begin
   if Wing < 0.6*Length then
      Factor := (1.0 + 0.037);
   else
      Factor := (1.0 + 0.048);
   end if;
end;
```

RESERVED WORDS IN LOWER CASE

begin   else   end   if

INSTRUCTOR NOTES

POINT OUT THAT SINCE RESERVED WORDS ARE RECOMMENDED TO BE IN LOWER CASE,

```
if CAN_FIT_ON_GRAPH then
else DRAW_CIRCLE;
  ...
end if;
```

LOOKS STRANGE AND IS HARD TO READ. THE EYE MUST LOOK UP AFTER READING THE "if" TO READ THE CAN_FIT_ON_GRAPH.

```
if Can_Fit_on_Graph then
   Draw_Circle;
else
  ...
end if;
```

IS MUCH EASIER TO READ.

USER SUPPLIED IDENTIFIERS

VERSION 1

```
if CAN_FIT_ON_GRAPH then
   DRAW_CIRCLE;
else
   DISPLAY_WARNING;
end if;
```

VERSION 2

```
if Can_Fit_on_Graph then
   Draw_Circle;
else
   Display_Warning;
end if;
```

CAPITALIZE FIRST LETTER OF EACH WORD IN IDENTIFIERS THAT ARE NOT RESERVED WORDS, UNLESS THE WORD IS A PREPOSITION.

INSTRUCTOR NOTES

Io (WITH LOWER CASE O) IS NOT THE CONVENTIONAL WAY OF WRITING ABOUT INPUT/OUTPUT IN TEXTS. IO (WITH CAPITAL O) IS THE CONVENTION.

AN EXCEPTION

VERSION 1

```
with Text_Io; use Text_Io
procedure Do_Something is
   ...
   package Color_Io is new ... ;
begin -- Do_Something
   ...
end Do_Something;
```

VERSION 2

```
with Text_IO; use Text_IO;
procedure Do_Something is
   ...
   package Color_IO is new ... ;
   ...
begin -- Do Something
   ...
end Do_Something;
```

CAPITALIZE THE IO IN IO PACKAGES

INSTRUCTOR NOTES

WHICH BRINGS US TO SUBPROGRAMS. IT IS EASIER TO DISTINGUISH DECLARATIONS FROM EXECUTABLE CODE WHEN SUBPROGRAMS ARE ALIGNED AS RECOMMENDED.

SUBPROGRAM STRUCTURE

VERSION 1

```
with Text_IO; use Text_IO;
   procedure Do_Something is
          ...
       begin
          ...
end Do_Something;
```

VERSION 2

```
with Text_IO; use Text_IO;
procedure Do_Something is
   ...
begin -- Do_Something
   ...
end Do_Something;
```

ALIGN with, function / procedure, begin, and end

INSTRUCTOR NOTES

IT HELPS TO KNOW WHAT THE BEGIN REFERS TO AS

- SUBPROGRAMS CAN BE NESTED
- BLOCKS HAVE A BEGIN

IT HELPS TO KNOW WHAT IS BEING ENDED AS

- SUBPROGRAMS CAN BE NESTED
- BLOCKS HAVE AN END

THIS CONVENTION HELPS IDENTIFY OVERALL STRUCTURE

SUBPROGRAM NAME

VERSION 1

```
with Text_IO; use Text_IO;
procedure Do_Something (Parameter : Some_Type) is
   ...
begin
   ...
end;
```

VERSION 2

```
with Text_IO; use Text_IO;
procedure Do_Something (Parameter : Some_Type) is
   ...
begin -- Do_Something
   ...
end Do_Something;
```

> COMMENT OUT THE SUBPROGRAM NAME AFTER THE begin.
>
> INCLUDE SUBPROGRAM NAME AFTER THE end.

WHICH BRINGS US TO PARAMETERS.

THE FIRST WAY OF WRITING THE SPECIFICATION IS TOO LONG AND MAY EXTEND OVER THE LINE.

THE SECOND METHOD PROVIDES NO QUICK WAY TO GRASP THE TWO PARAMETERS.

MULTIPLE PARAMETERS

VERSION 1

```
-- may extend over the line
procedure Do_Something (Parameter_1 : out Integer; Parameter_2 : out Boolean) is
   ...
end Do_Something;
```

OR

VERSION 2

```
-- not easy to grasp number of parameters
procedure Do_Something (Parameter_1 : out Integer;
   Parameter_2 : out Boolean) is
   ...
end Do_Something;
```

POINT OUT THE ALIGNMENT.

IT IS MUCH EASIER TO GRASP THE TWO PARAMETERS WHEN WRITTEN THIS WAY.

MULTIPLE PARAMETERS

VERSION 3

```
procedure Do_Something (Parameter_1 : out Integer;
                        Parameter_2 : out Boolean) is
   ...
begin -- Do_Something
   ...
end Do_Something;
```

INSTRUCTOR NOTES

THIS IS A GUIDELINE ONLY. DIFFERENT SHOPS MAY HAVE THEIR OUR GUIDELINE.

GUIDELINES

UNLESS MULTIPLE PARAMETERS CAN COMFORTABLY FIT ON ONE LINE, PLACE THEM ON DIFFERENT LINES AND ALIGN THE COLONS SEPARATING THE PARAMETER FROM ITS MODE (OR TYPE).

INSTRUCTOR NOTES

LONG PARAMETER NAMES

VERSION 1

```
procedure Do_Something (Parameter_1 : out Integer;
    Parameter_2_With_Very_Long_Name : out Boolean) is
    ...
begin -- Do_Something
    ...
end Do_Something;
```

INSTRUCTOR NOTES

EITHER OF THESE VERSIONS IS ACCEPTABLE. IF THE PARAMETER WITH THE LONG NAME IS WRITTEN FIRST, IT PROBABLY WOULD HAVE TO BE WRITTEN ON THE NEXT LINE.

LONG PARAMETER NAMES

VERSION 2

```
procedure Do_Something
   (Parameter_1                     : out Integer
    Parameter_2_With_Very_Long_Name : out Boolean) is
   ...
begin -- Do_Something
   ...
end Do_Something;
```

INSTRUCTOR NOTES

POINT OUT THAT THE RETURN IS OUTDENTED.

ADDING THE RETURN

```
function Function_Name (Parameter_1           : Boolean;
                        Parameter_2_Long_Name : Character)
                     return Boolean is
   ...
begin -- Function_Name
   ...
end Function_Name;
```

INSTRUCTOR NOTES

WHEN FORCED TO GO ON THE NEXT LINE, STILL OUTDENT THE RETURN.

ADDING THE RETURN AND EVEN LONGER PARAMETER NAMES

```
function Function_Name
   (Parameter_1                                      : Boolean;
    An_Even_Longer_Parameter_Name_for_Parameter_2 : Character)
   return Boolean is
      ...
begin -- Function_Name
      ...
end Function_Name;
```

POINT OUT THAT READABILITY AIDS UNDERSTANDING AND THAT UNDERSTANDING IS THE KEY TO RELIABILITY.

GUIDELINE

WHATEVER YOU DO, MAKE IT READABLE

INSTRUCTOR NOTES

FOLLOWING THIS CONVENTION ELIMINATES THE GUESS WORK IN THE DEBUGGING PHASE.

PARAMETER MODES

- EXPLICITLY STATE MODE in FOR PROCEDURES
- OMIT in FOR FUNCTIONS

INSTRUCTOR NOTES

POINT OUT "then" IS ON SAME LINE WITH THE if.

POINT OUT THE RESERVED WORDS, if, elsif, else, AND end ALIGN.

IF STATEMENTS

```
if Condition_1 then
   Some_Statements;
elsif Condition_2 then
   Some_Other_Statements;
else
   Something_Else_To_Do;
end if;
```

INSTRUCTOR NOTES

BREAK A COMPLEX BOOLEAN EXPRESSION AT A LOGICAL POINT.

IF with LONG CONDITION

```
if Condition_1 and
   A_Very_Long_Condition_2 then
      ...
end if;
```

INSTRUCTOR NOTES

POINT OUT THAT <u>case</u> AND <u>end</u> ALIGN.

POINT OUT THE <u>whens</u> ALIGN AS DO STATEMENTS FOR EACH ALTERNATIVE.

ARROWS MAY OR MAY NOT ALIGN.

CASE STATEMENTS

```
case Number_of_Choices is
   when 1 .. 10 =>
      Action_1;
   when 30      =>
      Action_2;
   when others  =>
      null;
end case;
```

- A SPACE ON EITHER SIDE OF THE => IMPROVES READABILITY

INSTRUCTOR NOTES

loop AND end ALWAYS ALIGN.

LOOP STATEMENTS

1.

```
for Loop_Index in Some_Range
loop
   ...
end loop;
```

2.

```
while Condition
loop
   ...
end loop
```

3.

```
loop
   ...
end loop;
```

INSTRUCTOR NOTES

THE LOOP NAME IS OUTDENTED AND WRITTEN IN ALL CAPITAL LETTERS. BOTH THESE ACTIONS FACILITATE SPOTTING THE LOOP WITHIN THE CODE.

NAMED LOOPS

```
   Some_Statement;
   Another_Statement:
LOOP_NAME:
   for Loop_Index in Some_Range
   loop
      Action_1;
      Action_2;
   end loop LOOP_NAME;
```

INSTRUCTOR NOTES

ASK THE CLASS "HOW MANY TYPES, OBJECTS, ETC.?"

DIFFICULT TO SEE THE FOREST FROM THE TREES.

MUST SEARCH FOR THE OBJECTS.

MUST SEARCH FOR THE IO.

DECLARATIVE PARTS

VERSION 1

```
procedure Do_Something is
   type Some_Type_1 is ... ;
   type Some_Type_2 is ... ;
   Object_1 : Some_Type_1;
   type Some_Type_3 is ... ;
   Object_2 : Some_Type_2;

   procedure Do_It is separate;
   package IO_on_Some_Type_1 is new ... ;
   procedure Another_Do_It is separate;
   package IO_on_Some_Type_2 is new ... ;

begin -- Do_Something
   ...
end Do_Something;
```

INSTRUCTOR NOTES

IT'S MUCH EASIER TO SEE THAT THERE ARE 1) THREE (3) TYPES DECLARED, 2) NO OBJECTS FOR THE THIRD TYPE ARE DECLARED, 3) IO CAN ONLY BE PERFORMED ON OBJECTS OF THE FIRST TWO TYPES AND 4) THERE ARE TWO (2) PROCEDURES STUBBED OUT.

DECLARATIVE PARTS

<u>VERSION 2</u>

```
procedure Do_Something is
   type Some_Type_1 is ... ;
   type Some_Type_2 is ... ;
   type Some_Type_3 is ... ;

   Object_1 : Some_Type_1;
   Object_2 : Some_Type_2;

   package IO_on_Some_Type_1 is new ... ;
   package IO_on_Some_Type_2 is new ... ;

   procedure Do_It is separate;
   procedure Another_Do_It is separate;

begin -- Do_Something
   ...
end Do_Something;
```

INSTRUCTOR NOTES

THIS IS A GENERAL GUIDELINE ONLY. FOR INSTANCE IT DOES NOT TAKE INTO ACCOUNT NAMED NUMBERS. THE POINT IS TO CLUSTER SIMILAR DECLARATIONS TOGETHER FOLLOWING THE RESTRICTIONS IMPOSED BY THE LANGUAGE REFERENCE MANUAL.

GUIDELINES

- LIST BY KIND OF DECLARATION
  - LIST TYPES TOGETHER
    - LIST OBJECTS TOGETHER
      - LIST INSTANTIATIONS TOGETHER
        - LIST BODIES TOGETHER

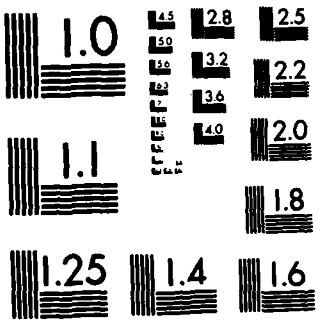
- USE BLANK LINE BETWEEN THE GROUPS

- ALIGN THE COLON FOR OBJECT DECLARATIONS

NOTE: THE LRM DOES PLACE SOME RESTRICTIONS!

AD-A143 501 ADA (TRADEMARK) TRAINING CURRICULUM PROGRAMMING METHODOLOGY M203 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA JUL 84 DAAB07-83-C-K514 5/8

UNCLASSIFIED F/G 5/9 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

INSTRUCTOR NOTES

AGAIN - USE OF SPACES ENHANCES READABILITY.

SPACES

- PLACE SPACE BEFORE PARENTHESIS IN FORMAL PARAMETER LISTS

```
procedure Interchange (Value_1, Value_2 : in out Integer);
```

- PLACE SPACE BEFORE AND AFTER ".."

```
for Loop_Index in 1 .. 10
loop
   ...
end loop;
```

INSTRUCTOR NOTES

AGAIN -- READABILITY.

SPACES (CONTINUED)

- PLACE A SPACE BEFORE AND AFTER EACH STATEMENT LABEL

  « The_Devil »

- PLACE SPACE BEFORE AND AFTER => IN
  - NAMED NOTATION
  - CASE STATEMENTS
  - EXCEPTION HANDLERS
  - ETC.

INSTRUCTOR NOTES

AGAIN -- READABILITY

SPACES (CONTINUED)

- PLACE SPACE BEFORE AND AFTER OPERATORS AND SPECIAL CHARACTERS EXCEPT FOR THE DIVISION OPERATOR WHEN THE OBJECTS CONSIST OF ONE (1) CHARACTER.

```
a + b        a rem b        e & f        X(1) / Y(1)
                      but
                      X/Y
```

- NO SPACE WHEN USING DOT NOTATION*

```
Record_Object.Selected_Component
Package_Name.Specific_Resource
```

*THIS IS A RESTRICTION OF THE LANGUAGE.

INSTRUCTOR NOTES

REMEMBER ...

- EACH SHOP HAS ITS OWN STANDARDS. IT IS YOUR RESPONSIBILITY TO FOLLOW THEM

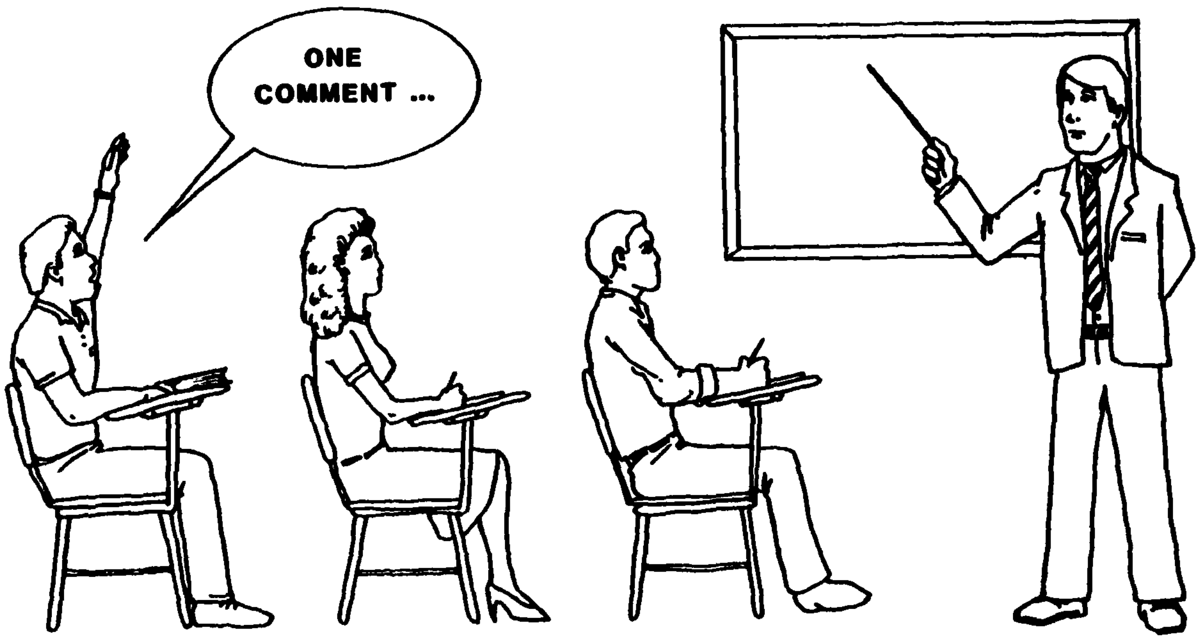
- LRM IMPLIES A CONVENTION

- END GOAL IS READABILITY. WHAT DO YOU WANT TO SEE?

YOU WILL BE THE READER TOO.

INSTRUCTOR NOTES

THIS SUBSECTION DEALS WITH COMMENTING CONVENTIONS.



VG 817

INSTRUCTOR NOTES

IF THE CODE IS IN ERROR, NO AMOUNT OF COMMENTING WILL HELP.

MAIN GOALS OF COMMENTS

- COMMENTS
  - ENHANCE READABILITY
  - HELP TEAM WORK TOGETHER
- NO AMOUNT OF COMMENTING CAN REPLACE WELL EXPRESSED STATEMENTS
- TOO MUCH COMMENTING CAN BE HARMFUL

INSTRUCTOR NOTES

WHAT IS THIS CODE DOING? A COMMENT OR TWO WOULD HELP.

NO COMMENTS

```
for I in 1 .. N
loop
   for J in 1 .. N
      loop
         X(I, J) := (I/J)*(J/I);
      end loop;
end loop;
```

WHAT IS THIS CODE DOING?

INSTRUCTOR NOTES

CAN'T FIND THE CODE THROUGH THE COMMENTS!!

EXCESSIVE COMMENTING

```
--  CURRENT COMPUTING PROGRAM
--  INPUT VALUES FOR RESISTANCE, FREQUENCY, AND INDUCTANCE
Get_Values_For (RESISTANCE, FREQUENCY, INDUCTANCE);
--  OUTPUT VALUES FOR RESISTANCE, FREQUENCY, AND INDUCTANCE
Output_Values_For (Resistance, Frequency, Inductance);
--  INPUT STARTING AND TERMINATING VALUES OF CAPACITANCE
Get (Starting_Point);
Get (Terminating_Point);
--  SET CAPACITANCE TO STARTING VALUE
Capacitance := Starting_Point;
--  SET INITIAL VALUE OF VOLTAGE TO 1.0
Voltage := 1.0;
--  PRINT VALUE OF VOLTAGE
--  COMPUTE CURRENT VALUE OF A
A := ... ;
-- ETC.
```

INSTRUCTOR NOTES

DO NOT MAKE A PRONOUNCEMENT AS TO HOW MANY COMMENTS PER LINES OF CODE.

MORAL

RIGHT AMOUNT OF COMMENTING
USUALLY LIES BETWEEN
THESE EXTREMES

INSTRUCTOR NOTES

A COMMENT IS OF ZERO VALUE IF IT IS WRONG. HERE THE ERROR IS SUFFICIENTLY OBVIOUS THAT IT IS NOT LIKELY TO BE MISLEADING. BUT IN MANY CASES THE COMMENT CAN BE MISLEADING.

MISLEADING COMMENT

```
--  TEST FOR NEGATIVE VALUE FOR X
if (X-1) < 0 then
   ...
else
   ...
end if;
```

REGARDLESS OF HOW MANY COMMENTS YOU WRITE, MAKE SURE THAT THE COMMENT AND CODE AGREE.

INSTRUCTOR NOTES

Odd_Number ENCOURAGES US TO BELIEVE THE COMMENT. WE SEE THE COMMENT, WE SEE USE OF Odd_Number SO WE <u>DO</u> <u>NOT</u> CHECK THE CODE.

MISLEADING OR WRONG?

```
--  TESTING THIS TIME FOR
--  ODD NUMBERS
if Number Mod 2 = 0 then
   Sum := Sum + Number;
   Odd_Number := Odd_Number + 1;
end if;
```

INSTRUCTOR NOTES

MORAL

MAKE SURE COMMENTS
AND CODE AGREE

INSTRUCTOR NOTES

YES, TIME IS USUALLY CRITICAL. BUT DON'T YOU WANT USEFUL COMMENTS IN THE CODE YOU MUST MAINTAIN?

REMEMBER

IF THE CODE IS CORRECT, THEN <u>CHANGE THE COMMENT</u>!

INSTRUCTOR NOTES

HERE AGAIN, COMMENT AND CODE DISAGREE. WHY IS THE TEST AGAINST 3.01 INSTEAD OF 3.0? PROBABLY BECAUSE OF ROUNDING BUT WE CAN'T ASSUME THAT IS THE CASE.

RIGHT OR WRONG?

```
E := E + 0.5
--  TEST FOR VOLTAGE EXCEEDING 3.0
if E > 3.01 then
    ...
end if;
```

IF THIS IS CORRECT, EXPLAIN IT! IF IT REFLECTS A POOR ALGORITHM, CHANGE THE ALGORITHM.

INSTRUCTOR NOTES

THE EXTRA TIME NEEDED TO REWRITE IS WELL WORTH IT.

MORAL

DON'T COMMENT BAD CODE --

REWRITE IT

INSTRUCTOR NOTES

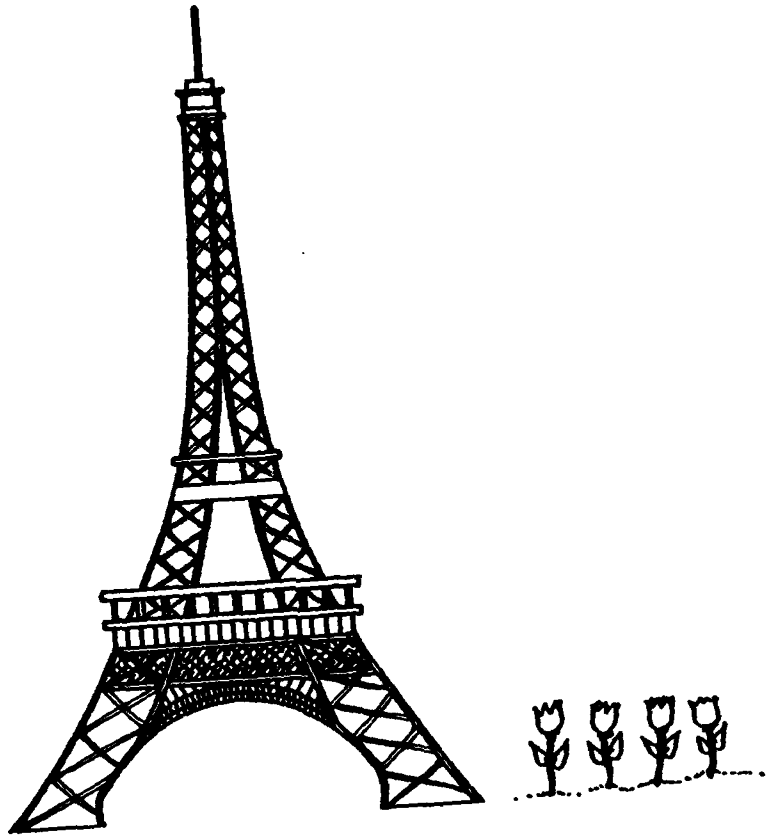
ASSUME THIS IS IN A PACKAGE BODY. ASSUME KNOWLEDGE OF List (AN ARRAY OF RECORDS) AND Length (THE LENGTH OF THE LIST).

HOW'S THIS?

```
--
--  PROCEDURE DELETE REMOVES A NAME FROM THE
--  DIRECTORY IF THE NAME EXISTS IN THE DIRECTORY.
--  THE DIRECTORY IS THEN REORGANIZED SO THAT
--  THE SLOT IS REMOVED.  IF THE NAME IS NOT
--  IN THE DIRECTORY, THE EXCEPTION Not_Found
--  IS RAISED.
--
procedure Delete (Name : in Name_String) is
   Index : Integer range 0 .. 100;
begin -- Delete
   Find (Name, Index);              -- returns 0 if name not present, otherwise position
   if Index /= 0 then               --  is name present
      List (1 .. Length-1)          -- yes, reorganize
            := List (1 .. Index_1) & List (Index + 1 .. Length);
   else
      raise Not_Found;              --  no
   end if;
end Delete;
```

INSTRUCTOR NOTES

THIS SUBSECTION DEALS WITH NAMING CONVENTIONS.



Je m' applle ...

VG 817

INSTRUCTOR NOTES

ALL POINTS ARE EQUALLY IMPORTANT.

NAMING CONVENTIONS

- GOOD NAMING CONVENTIONS CAN ENHANCE PROGRAM READABILITY
- GOOD NAMING CONVENTIONS AID IN THE PRODUCTION OF SELF DOCUMENTING CODE
- GOOD NAMING CONVENTIONS HELP MODEL THE PROBLEM DOMAIN

INSTRUCTOR NOTES

- WHAT ARE THE REQUIREMENTS FOR GOOD IDENTIFIERS IN A LARGE ADA PROGRAM?
- THERE ARE PROBLEMS IN SELECTING GOOD IDENTIFIERS
- THIS SUBSECTION PROVIDES A FEW GUIDELINES FOR SELECTING GOOD IDENTIFIERS

NAMING CONVENTIONS

THIS SUBJECT DESERVES A GREAT DEAL OF ATTENTION. ADA'S GOAL IS TO PROVIDE MAINTAINABLE CODE. ONE PRIME INGREDIENT OF MAINTAINABLE CODE IS READABILITY. NAMES MUST BE READABLE IN ORDER FOR THE CODE TO BE MAINTAINABLE.

BUT WE DO NEED GUIDELINES TO CONSTRAIN THE CHOICES PROGRAMMERS MAKE. THE PROGRAMMER NEEDS TO CONCENTRATE ON MORE FUNDAMENTAL ISSUES.

Ada DESIGN METHODS TRAINING SUPPORT CASE STUDIES REPORT DECEMBER 1983 "Guidelines for the Selection of Identifiers."

INSTRUCTOR NOTES

THE POINT OF THE VIEWGRAPH IS TO IMPRESS UPON THE STUDENT THE PLETHORA OF ITEMS TO BE NAMED. DO NOT GET BOGGED DOWN IN ANY DISCUSSION OF A SPECIFIC ENTITY.

A LIST OF ENTITIES TO BE NAMED

OBJECTS	ENTRIES
NAMED NUMBERS	FORMAL PARAMETERS
TYPES	GENERICS
SUBTYPES	INSTANTIATIONS
NON CHARACTER ENUMERATION LITERALS	EXCEPTIONS
RECORD COMPONENTS (DISCRIMINANTS)	LOOP PARAMETER
PACKAGES	NAMED LOOPS
SUBPROGRAMS	NAMED BLOCKS
TASKS	STATEMENT LABELS

INSTRUCTOR NOTES

USE THIS FOIL TO SET THE STAGE FOR THE REMAINDER OF THE SECTION.

NAMES

- ADA PLACES NO LIMIT ON IDENTIFIER LENGTH

- USE APPROPRIATE NAMES

TOO CRYPTIC	BETTER
X	Value_1
N	Number
Seg	Segment
Cvt_To_ACP	Convert_To_ACP

INSTRUCTOR NOTES

IS Log_Line AN ABBREVIATION FOR Logical_Line OR IS IT AN UNABBREVIATED VERB? OR, MORE OBSCURE, IS IT LOGARITHM?

IS CAT AN ABBREVIATION FOR CATENATE, CATALOG, OR CATEGORY?

Ret_Message COULD BE Return_Message, Retarget_Message, or Retry_Message?

IS SVC AN ABBREVIATION FOR SERVICE OR DOES IT STAND FOR SUPERVISOR CALL?

CONFUSION

- ABBREVIATIONS ARE MUCH LESS OBVIOUS TO THE READER THAN TO THE PERSON WHO DEVISED THEM.

  Log_Line

  Cat

  Ret_Message

  SVC

INSTRUCTOR NOTES

CPU => CENTRAL PROCESSING UNIT

IO => INPUT OUTPUT

JANAP
ACP } SPECIFIC MESSAGE FORMATS

MCB => MESSAGE CONTROL BLOCK

RI => ROUTING INDICATOR

Sin => SINE

Cos => COSINE

Tan => TANGENT

Atan => Arc Tangent

POINT OUT USE OF ALL CAPITAL VERSUS FIRST LETTER CAP.

ACCEPTABLE ABBREVIATIONS

- STANDARD
  - CPU
  - IO
- APPLICATION SPECIFIC
  - JANAP
  - ACP
  - MCB
  - RI
- APPLICATION SPECIFIC
  - SIN
  - COS
  - TAN
  - ATAN

INSTRUCTOR NOTES

- FIRST BULLET

  NUM, NO, AND NUMBER ARE THREE (3) WAYS OF WRITING NUMBER

- SECOND BULLET

  MST FOR MOST, LST FOR LEAST, AND LGTH FOR LENGTH VIOLATE THIS RULE

- THIRD BULLET

  LOG FOR LOG OR LOGARITHM

  SCTY FOR SECURITY OR SECRETARY

GUIDELINES FOR ABBREVIATIONS

- A CONSISTENT WAY TO WRITE AN ABBREVIATION

  Num_RIs   Ext_Serial_No   Segment_Number

- DO NOT ABBREVIATE UNLESS IT SAVES AT LEAST THREE (3) LETTERS

  MST   LST   LGTH

- IT SHOULD NOT BE POSSIBLE TO MISTAKE THE ABBREVIATION OF ONE WORD FOR THE ABBREVIATION OR FULL FORM OF ANOTHER WORD.

  Log   Scty

INSTRUCTOR NOTES

THE IMPORTANT POINT IS TO BE CONSISTENT ON A PROJECT. A PROJECT MAY WANT A DICTIONARY OF STANDARD ABBREVIATIONS.

HOW TO ABBREVIATE

- TRUNCATION

  Segment => Seg

- DROP VOWELS

  Segment => Sgmnt

$\boxed{\text{BE CONSISTENT}}$

INSTRUCTOR NOTES

ASK CLASS WHICH THEY PREFER. STRESS THAT THE GOAL IS READABILITY.

MAKING PLURALS

- WRITE RIS AS RIs

  RIS APPEARS TO BE TALKING ABOUT A RIS

  RIs IS CLEARLY ROUTING INDICATORS

- BENEFIT IS MORE APPARENT WHEN APPEARING AS PART OF ANOTHER IDENTIFIER

  Bad_RIS or Bad_RIs

  Check_RIS or Check_RIs

  Remove_IDS or Remove_IDs

INSTRUCTOR NOTES

NOUNS ARE OPERATED ON BY VERBS. OBJECTS ARE OPERATED ON BY OPERATORS OR OPERATIONS.

NAMING OBJECTS

OBJECTS SHOULD BE NAMED WITH NOUNS

```
Balance : Money_Type;
Message : Message_Record_Type;
Security_Table : array (Security_Code) of Security_Type;
Target : Target_Record_Type;
```

INSTRUCTOR NOTES

A STATEMENT IN ADA PERFORMS AN ACTION. ACTIONS ARE ASSOCIATED WITH VERBS.

NAMING PROCEDURES

PROCEDURES SHOULD BE NAMED WITH VERBS

```
procedure Check_RIs (Message : Message_Type);
```

AS A PROCEDURE CALL IS A STATEMENT

```
Check_RIs (Incoming_Message);
```

INSTRUCTOR NOTES

THE FOIL IS SELF EXPLANATORY.

NAMING FUNCTIONS

- NAME FUNCTIONS WITH A NOUN OR CONDITIONAL CLAUSE

```
function End_of_File (File : File_Type) return Boolean;
function Mean_of (List : List_Type) return Float;
```

AS A FUNCTION CALL IS USED WITHIN ...

- A CONDITION

```
while End_of_File (Input_File)
loop
   ...
end loop;
```

- OR AN EXPRESSION

```
Mean := Mean_of (List_of_Grades);
```

INSTRUCTOR NOTES

Address_Access_Type IS A POINTER TO ANOTHER RECORD TYPE CONTAINING COMPONENTS FOR STREET, CITY, STATE, AND ZIP CODE.

THIS AGAIN IS A DEBUGGING AID. WHEN SEEN IN AN EXPRESSION IT IS CLEAR THAT IT IS A COMPONENT OF A RECORD. READABILITY IS THE KEY THEME.

RECORD COMPONENTS

- SUFFIX COMPONENT NAME WITH "_Part"

- IF WE HAVE

```
type Person_Record_Type is
record
   Last_Name_Part      : String (1 .. 20);
   First_Name_Part     : String (1 .. 10);
   Middle_Initial_Part : Character;
   Address_Part        : Address_Access_Type;
end record;

type Mailing_List_Type is array (1 .. 100) of Person_Record_Type;
Mailing_List : Mailing_List_Type;
```

- THEN WE CAN WRITE

```
Mailing_List(3).Address_Part.Zip_Code_Part
```

INSTRUCTOR NOTES

HAVING CODE READ LIKE AN ENGLISH SENTENCE FORCES THE PROGRAMMERS VIEW OF THE PROBLEM ONTO PAPER AND THEREFORE HELPS PROVIDE SELF DOCUMENTING CODE.

THE END GOAL

```
Push (Element => Name, On_To => This_Stack);
while End_of_File (Input_File) ... ;
Ada_101_Mean := Mean_of (Ada_101_Class_Grades);
```

GET THE CONCEPTUAL VIEW OF

THE PROBLEM INTO THE CODE

INSTRUCTOR NOTES

1. Current_Balance or Balance

2. Account_Number (ONLY ABBREVIATE NUMBER IF THERE IS A LIST OF APPROVED ABBREVIATIONS)

3. Deposit_Money_Into

4. Is_Empty

5. Current_Balance_Of

AN EXERCISE

WRITE IDENTIFIERS FOR THE FOLLOWING:

1. A VARIABLE TO REPRESENT THE CURRENT BALANCE OF A CHECKING ACCOUNT

2. A VARIABLE TO REPRESENT THE ACCOUNT NUMBER

3. A PROCEDURE TO BE USED TO DEPOSIT MONEY INTO THE ACCOUNT

4. A FUNCTION TO DETERMINE WHETHER THE ACCOUNT IS EMPTY

5. A FUNCTION TO DETERMINE THE CURRENT BALANCE

INSTRUCTOR NOTES

ALLOCATE 160 MINUTES FOR THIS SECTION.

Section 4
ENSURING RELIABILITY

INSTRUCTOR NOTES

REVIEW THE POINTS THAT WERE MADE AT THE BEGINNING OF THE COURSE THAT THE PROGRAMMER HAS AN OBLIGATION TO ENSURE THAT HIS OR HER PROGRAMS ARE CORRECT.

MENTION AGAIN THE DIJKSTRA QUOTE ABOUT TESTING SHOWING THE PRESENCE OF BUGS BUT NEVER THEIR ABSENCE.

ENSURING RELIABILITY

- AS WE SAID BEFORE, EVERY PROGRAMMER HAS AN OBLIGATION TO ENSURE THAT HIS OR HER PROGRAMS ARE CORRECT!
- RELIABILITY CAN'T BE TESTED IN
- IT MUST BE BUILT IN
- AND THE ONE WHO MUST DO IT IS THE PROGRAMMER WHO ORIGINALLY CREATES THE PROGRAM

INSTRUCTOR NOTES

REVIEW THE REASONS WHY RELIABILITY IS IMPORTANT. FOLLOW THE ARGUMENT GIVEN IN SECTION 1 OF THE COURSE.

MENTION WEAPONS SYSTEMS AS AN EXAMPLE OF A SYSTEM WHERE PROGRAM RELIABILITY IS VITAL TO BOTH HUMAN SAFETY AND NATIONAL SECURITY.

ON THE LAST POINT, TRY TO BRING THE POINT HOME BY MAKING IT PERSONALLY IMPORTANT TO EACH INDIVIDUAL.

WHY RELIABILITY IS IMPORTANT

- RELIABILITY IS OF VITAL IMPORTANCE IN VIRTUALLY ALL SYSTEMS
- AT THE LEAST, UNRELIABLE PROGRAMS COST EXTRA MONEY AND TIME TO DEVELOP, DEBUG, AND PUT INTO OPERATION
- AT THE EXTREMES, UNRELIABILITY CAN COST LIVES OR CAN ADVERSELY AFFECT NATIONAL SECURITY
- RELIABILITY IS IMPORTANT TO A PROGRAMMING ORGANIZATION BECAUSE IT AFFECTS THE CUSTOMER SATISFACTION WITH THEIR PRODUCT
- FINALLY, RELIABILITY IS IMPORTANT TO THE INDIVIDUAL PROGRAMMER BECAUSE IT HELPS MAINTAIN A GOOD SELF IMAGE AND JOB SATISFACTION

INSTRUCTOR NOTES

READ THE ORIGINAL OF THIS STORY IN WEINBERG'S BOOK (Pg. 17-19). A COPY OF THOSE PAGES IS PROVIDED ON THE FOLLOWING PAGES (a, b and c)..

THE PURPOSE OF THESE THREE SLIDES IS TO INJECT A LITTLE HUMOR BEFORE WE GET TO THE DRYER TECHNICAL MATERIAL.

THE SLIDES SHOULD BE PUT UP AND THE STORY RELATED VERBALLY (AS IF FROM PERSONAL EXPERIENCE). YOU MIGHT WANT TO PUT IN SOME MORE DETAILS FROM THE ORIGINAL REFERENCED ABOVE.

THIS FIRST SLIDE SETS THE STAGE BY DESCRIBING A SITUATION THAT SHOULD FEEL FAMILIAR TO MANY STUDENTS.

—in which they are developed. Looking honestly at the situation, we are never looking for the best program, seldom looking for a good one, but always looking for one that meets the requirements.

SPECIFICATIONS

Of all the requirements that we might place on a program, first and foremost is that it be correct. In other words, it should give the correct outputs for each possible input. This is what we mean when we say that a program "works," and it is often and truly said that "any program that works is better than any program that doesn't."

An example may serve to drive home this point to those whose minds are tangled in questions of efficiency and other secondary matters. A programmer was once called to Detroit to aid in the debugging of a new program—one that was to determine the parts requirements to build a certain set of automobiles. The input to the program was a deck of cards, each card representing a purchase order for an automobile, with different punches representing the different options selected by the customer. The program embodied the specifications relating the various options to the parts that would be needed. For instance, the choice of upholstery for the rear seat might be determined by such factors as body color, body style, options for deluxe or leatherette upholstery, and whether or not the car was air conditioned. The air-conditioning option is a good example of the basic complexity of the problem, for though to an untrained eye the choice of air conditioning might have no connection with the choice of rear seat upholstery, it might very well require spaces for extra ducts. In general, then, each option might have some effect on the choice of parts made, so the determination of parts requirements was an excellent job for the computer.

Unfortunately, when this programmer arrived on the scene, the basic approach to the problem had long been settled—and settled badly. Each option—as it affected each choice—was reflected as an individually programmed test and branch in the program. In a way, the program was an enormous tree, with more than 5000 branches, representing the decisions leading to part selection. Cast in this form—and with 16 programmers working at the same time—it was impossible to debug, as each and every case had to be tested separately. To test the program, a particular card would be put in and the output would be observed. When our programmer arrived, things were so bad that typical cards were calling for the production of cars with eight tires, no engine, and three sets of upholstery. In short, a disaster.

As is usual with programming disasters, nobody recognized it as such.

Instead, the whole crew had gone on double shift to get out the bugs, and new programmers, including our hero, were brought in from all over the country. Naturally, this led to worse confusion than ever, and our programmer, after a few days, determined that it was hopeless business—and in any case not reason enough to be away from his family and working night and day. He was roundly condemned for his uncooperative attitude but was allowed to leave.

While on the plane, he had his first opportunity in a week to reflect calmly. He immediately saw the error in the approach and perceived that a much better approach would be to divide the work into two phases. The main operational program would simply loop through a set of specially constructed specifications tables, so that all decisions would be made with a single test reapplied to different parts of the table. In that way, the program was at least assured to produce the right number of tires, engines, and so forth. The tables themselves would be compiled from input written in essentially the form of the engineering specifications. This would allow the engineering personnel, rather than the programmers, to check the specifications, and also permit one part of the specification to be changed without changing all parts further down a decision tree.

By the time he got off the plane, he had coded the two programs. It was a day's work to check them out, and another two days' work with the local assembly plant engineers to create the specifications in input form. After a week's testing in the plant, he was about to return to notify Detroit of the news when he got a telegram saying that the project had been cancelled—since the program was impossible to write.

After a quick call and a plane trip, he was back in Detroit with his version of the program. A demonstration to the executives convinced them that the project could continue, and then he was asked to make a presentation to the rest of the programmers. Naturally, they were a rather cool audience—a phenomenon to which we shall return in our discussions—but they sat quietly enough through his explanation of the method. Even at the end, there was a lack of questioning—until the original creator of the old system raised his hand.

"And how long does *your* program take?" he asked—emphasizing the possessive.

"That varies with the input," was the reply, "but on the average, about ten seconds per card."

"Aha," was the triumphant reply. "But *my* program takes only one second per card."

The members of the audience—who had, after all, all contributed to the one-second version—seemed relieved. But our hero, who was rather young and naive, was not put down by this remark. Instead, he calmly observed, "But your program doesn't work. If the program doesn't have

to work, I can write one that takes one millisecond per card—and that's *faster* than our card reader."

This observation—though it undoubtedly failed to win our hero any friends—contains the fundamental truth upon which all programming evaluation must be based. If a program doesn't work, measures of efficiency, of adaptability, or of cost of production have no meaning. Still, we must be realistic and acknowledge that probably no perfect program was ever written. Every really large and significant program has "just one more bug." Thus, there are degrees of meeting specifications—of "working"—and evaluation of programs must take the type of imperfection into account.

Any compiler, for example, is going to have at least "pathological" programs which it will not compile correctly. What is pathological, however, depends to some extent on your point of view. If it happens in your program, you hardly classify it as pathological, even though thousands of *other* users have never encountered the bug. The producer of the compiler, however, must make some evaluation of the errors on the basis of the number of users who encounter them and how much cost they incur. This is not always done scientifically. Indeed, it often amounts to an evaluation of who shouts the loudest, or who writes to the highest executive. But whatever system is chosen, some bugs will remain, and some people will be unhappy with the same compiler that satisfies thousands.

In effect, then, there is a difference between a program written for one user and a piece of "software." When there are multiple users, there are multiple specifications. When there are multiple specifications, there are multiple definitions of when the program is working. In our discussions of programming practices, we are going to have to take into account the difference between programs developed for one user and programs developed for many. They will be evaluated differently, and they should be produced by different methods.

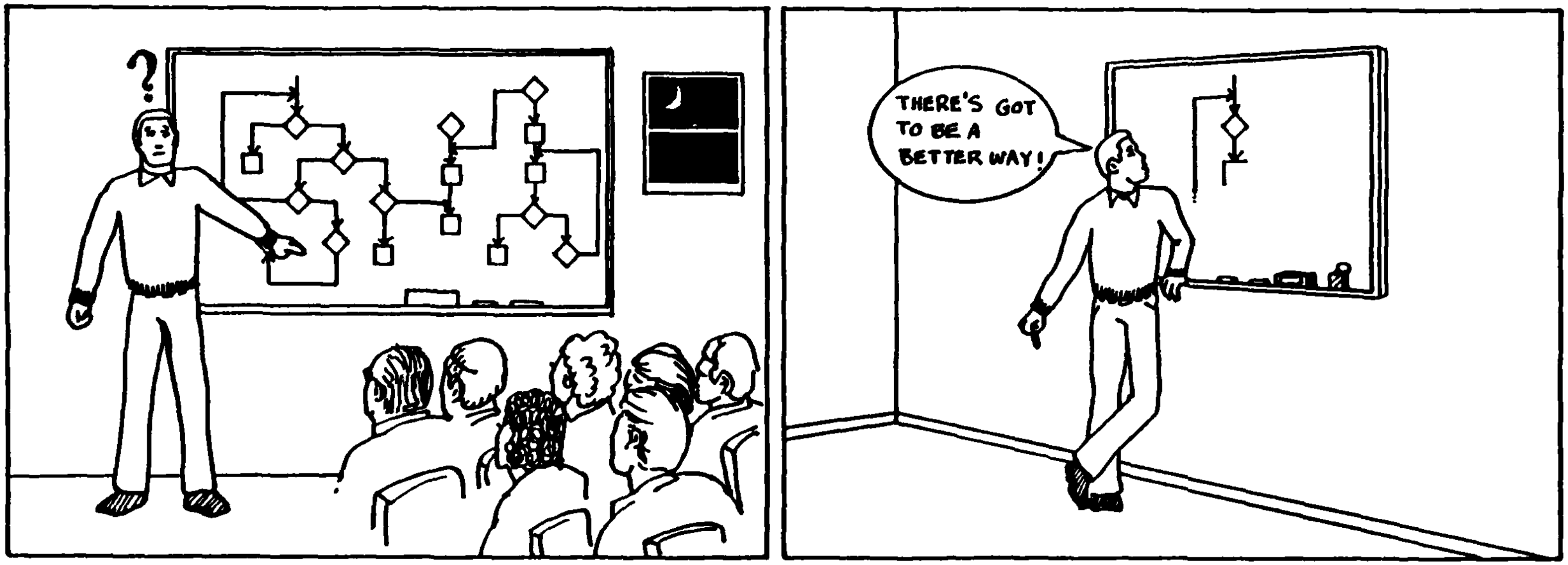
SCHEDULE

Even after questions of meeting specifications have been set aside, the question of efficiency is still not uppermost. One of the recurring problems in programming is meeting schedules, and a program that is late is often worthless. At the very least, we have to measure the costs of *not having* the program against any potential savings that a more efficient program would produce. In one noteworthy case, the customer of a software firm estimated that the linear programming code being developed would save more than one million dollars per month in the company's oil refining operations. Even one month's delay in schedule would result in a loss that

A PARABLE

ADAPTED FROM WEINBERG, "THE PSYCHOLOGY OF COMPUTER PROGRAMMING"

A PROGRAMMER WAS ONCE CALLED OUT TO A FIELD OFFICE LOCATION TO CONTRIBUTE TO AN AROUND-THE-CLOCK EFFORT TO TEST AND DEBUG (AS WE ALL KNOW, THAT REALLY MEANS FINISH *DESIGNING AND CODING*) A *LARGE PROGRAM INVOLVING THOUSANDS OF PROGRAMMED BRANCHES TO* HANDLE LARGE COMBINATIONS OF OPTIONS. AFTER GETTING FRUSTRATED WITH THE TASK, HE PLEADED FAMILY PROBLEMS AND LEFT FOR HOME. ON THE WAY BACK, HE SAW THE ESSENTIAL ERROR IN THE DESIGN OF THE PROGRAM. HE FOUND A WAY TO REPLACE THE THOUSANDS OF BRANCHES WITH A MUCH SIMPLER PROGRAM DRIVEN BY A DATA TABLE. BACK AT THE MAIN OFFICE, HE CODED UP THE PROGRAM AND WORKED WITH THE LOCAL ENGINEERS TO DEVELOP THE NECESSARY DRIVING TABLES.



INSTRUCTOR NOTES

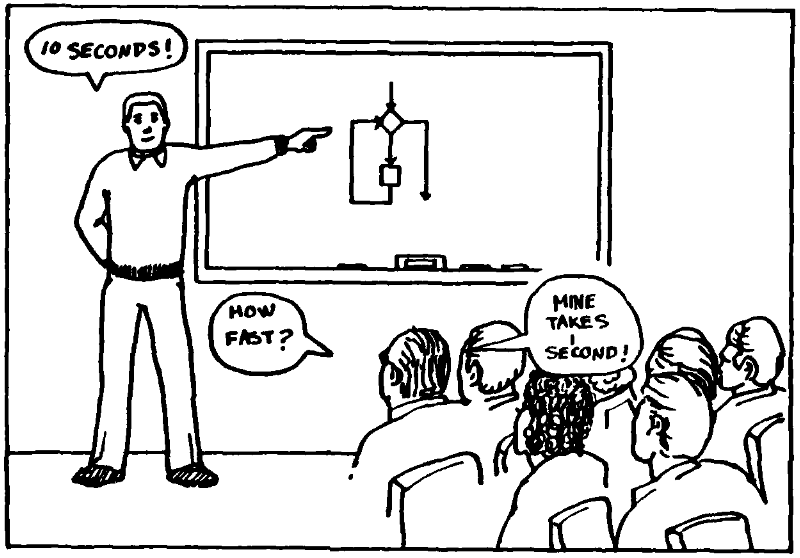
THIS SLIDE SETS UP THE POINT BY FOCUSING ON THE EFFICIENCY ISSUE THAT IS, DESPITE GOOD INTENTIONS, DEEPLY INGRAINED IN ALMOST ANYONE WHO HAS EVER DONE PROGRAMMING.

IT ALSO GETS AT THE EGO INVOLVEMENT IN A PROGRAMMING PROJECT THAT WE WANT TO SHOW HOW TO AVOID LATER IN THIS SECTION (EGOLESS PROGRAMMING). MAKE A POINT HERE OF POINTING OUT THE HOSTILE REACTION OF THE LOCAL PROGRAMMING STAFF.

A PARABLE

OUR PROGRAMMER RETURNED TO THE FIELD OFFICE TRIUMPHANTLY BEARING HIS PROGRAM, EXPECTING TO BE HAILED AS A CONQUERING HERO. INSTEAD (AS HE PROBABLY SHOULD HAVE REALIZED) HE WAS MET WITH A CHILLY RECEPTION FROM THE LOCAL STAFF. IN A PRESENTATION OF HIS APPROACH, HE WAS ASKED (BY THE DESIGNER OF THE ORIGINAL PROGRAM, OF COURSE) THE TELLING QUESTION: "AND TELL US, HOW FAST DOES YOUR PROGRAM RUN?"

HE REPLIED THAT IT WOULD TAKE ABOUT 10 SECONDS PER TRANSACTION. THE ORIGINAL QUESTIONER THEN MADE WHAT HE EXPECTED WOULD BE THE FATAL THRUST AT THE HERO'S PROGRAM: "WELL THAT PROVES YOUR APPROACH IS ALL WET. <u>MY</u> APPROACH ONLY TAKES 1 SECOND PER TRANSACTION!"



INSTRUCTOR NOTES

THIS IS THE PUNCH LINE OF THE STORY.

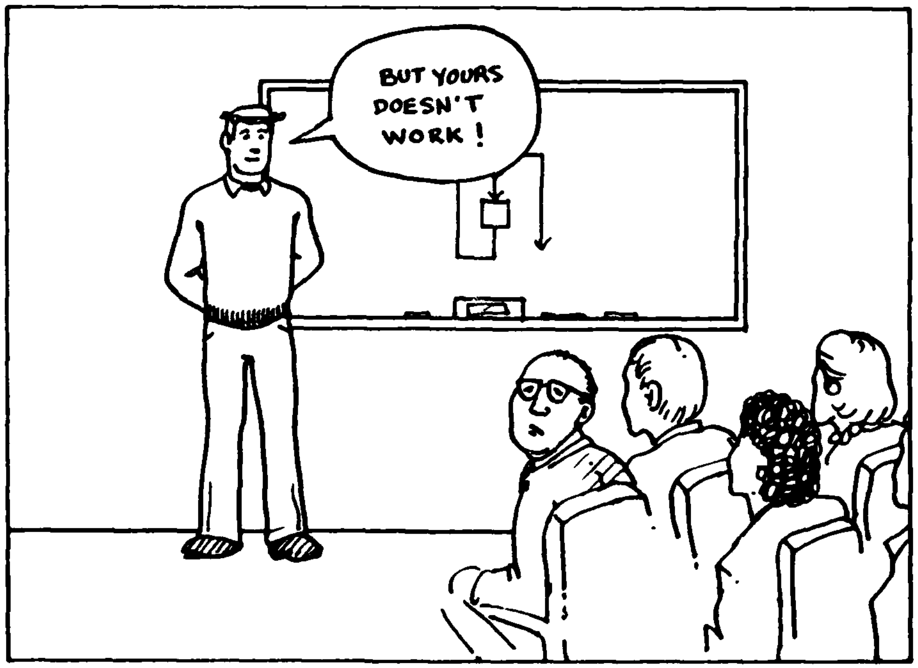
EMPHASIZE THE POINT THAT IN A PROGRAM RELIABILITY IS MORE IMPORTANT THAN SPEED.

YOU SHOULD SOFTEN THE POINT SOMEWHAT BY POINTING OUT THAT MANY PROGRAMS <u>DO</u> HAVE IMPORTANT THROUGHPUT OR RESPONSE TIME REQUIREMENTS THAT MUST NOT BE NEGLECTED, BUT THEY STILL TAKE SECOND PLACE TO RELIABILITY.

A PARABLE

BUT ULTIMATE DESTINY WAS WITH THE HERO. HE CALMLY PARRIED THE THRUST WITH THE COMMENT, "BUT YOUR PROGRAM DOESN'T WORK. I COULD MAKE A PROGRAM THAT WORKS AT A MILLISECOND PER TRANSACTION IF IT DOESN'T HAVE TO WORK."

THIS ENDED THE ARGUMENT, AND OUR HERO RECEIVED HIS WELL DESERVED ACCOLADES AFTER ALL.



INSTRUCTOR NOTES

ALL PARABLES HAVE A MORAL, SO HERE IT IS.

JUST SAY, "AND THE MORAL IS ..."

THE MORAL

IT DOESN'T MATTER HOW FAST YOUR PROGRAM RUNS IF
IT DOESN'T DO THE RIGHT THING!

INSTRUCTOR NOTES

NOW THAT THE STUDENTS ARE CONVINCED OF THE IMPORTANCE OF RELIABILITY, WE MOVE ON TO SHOW *HOW TO GO ABOUT ACHIEVING RELIABILITY.*

SAY IN A FEW WORDS HOW THE TECHNIQUES DISCUSSED PREVIOUSLY HELP IMPROVE RELIABILITY BY SIMPLIFYING OUR PROGRAMS.

HOW CAN RELIABILITY BE ACHIEVED?

- AS DISCUSSED BEFORE, RELIABILITY CAN'T BE TESTED IN:

  PROGRAM TESTING CAN SHOW THE PRESENCE OF BUGS, BUT NEVER THEIR ABSENCE

- THEREFORE WE MUST APPLY OUR HUMAN UNDERSTANDING TO SHOW THAT OUR PROGRAMS ARE CORRECT

- BECAUSE OF OUR HUMAN LIMITATIONS, WE MUST SIMPLIFY OUR PROGRAMS TO THE GREATEST POSSIBLE EXTENT

- THE LIMITATION OF CONTROL STRUCTURES TO ONE-INPUT, ONE-OUTPUT STRUCTURES (STRUCTURED PROGRAMMING) RESULTS IN SIMPLER PROGRAMS

INSTRUCTOR NOTES

NOW WE WILL TURN TO WAYS OF IMPROVING RELIABILITY BY IMPROVING OUR ABILITY TO UNDERSTAND THE PROGRAMS.

WARN THE STUDENTS ABOUT THE COMING THEORY, BUT EMPHASIZE THAT THE END RESULT IS A SET OF PRACTICAL TECHNIQUES THAT CAN BE APPLIED IN EVERYDAY PROGRAMMING.

CODING TECHNIQUES TO INCREASE RELIABILITY

- PRECEDING SECTIONS HAVE DISCUSSED TECHNIQUES FOR ENHANCING PROGRAM SIMPLICITY
- IN THIS SECTION WE WILL CONCENTRATE ON TECHNIQUES FOR IMPROVING OUR UNDERSTANDING OF OUR PROGRAMS
- THIS WILL THEN LEAD TO IMPROVED SOFTWARE RELIABILITY
- OUR GOAL IS TO DESCRIBE PRACTICAL TECHNIQUES THAT CAN BE APPLIED IN EVERYDAY PROGRAMMING
- IT IS NECESSARY, HOWEVER, TO PROVIDE SOME THEORETICAL BACKGROUND CONCERNING PROGRAM CORRECTNESS

INSTRUCTOR NOTES

IN THIS SECTION WE COVER WAYS OF ENHANCING OUR CONFIDENCE THAT OUR PROGRAM ARE CORRECT.

ENSURING RELIABILITY

DEMONSTRATION OF CORRECTNESS

INSTRUCTOR NOTES

ON THIS SLIDE, JUST POSE THE QUESTIONS. TRY TO GET THE STUDENTS TO LOOK INSIDE AND THINK ABOUT WHAT THEY THINK ABOUT WHEN THEY ARE PROGRAMMING.

DEMONSTRATION OF CORRECTNESS

- WHAT GOES THROUGH A PROGRAMMER'S MIND WHILE COMPOSING A PROGRAM?
- WHY IS ONE STATEMENT OR CONSTRUCT CHOSEN OVER ALL OTHER POSSIBILITIES?

INSTRUCTOR NOTES

HERE THE QUESTION POSED ON THE PREVIOUS SLIDE SHOULD BE ANSWERED.

A PROGRAMMER CHOOSES A STATEMENT BECAUSE HE HAS GONE THROUGH SOME KIND OF INTERNAL MENTAL EXERCISE THAT HAS CONVINCED HIM THAT THE STATEMENT IS THE RIGHT ONE.

LET THE LAST POINT REALLY SINK IN.

DEMONSTRATION OF CORRECTNESS

- THE PROGRAMMER CHOOSES A PARTICULAR STATEMENT BECAUSE IT IS THE <u>RIGHT</u> STATEMENT AT THAT POINT IN THE PROGRAM

- HE HAS CONVINCED HIMSELF THAT THE STATEMENT HE HAS CHOSEN IS CORRECT

- SO ...

  PROGRAMMING IS REALLY NOTHING MORE OR LESS THAN A DEMONSTRATION OF CORRECTNESS BY THE PROGRAMMER FOR HIS OWN BENEFIT

INSTRUCTOR NOTES

REALIZING WHAT GOES ON IN THE PROGRAMMER'S HEAD DURING PROGRAMMING POINTS THE WAY TO TWO WAYS OF IMPROVING CODE RELIABILITY.

DEMONSTRATION OF CORRECTNESS

- THE WAY TO ENHANCING CODE RELIABILITY IS THEN

  (1) TO EQUIP THE PROGRAMMER WITH TECHNIQUES TO IMPROVE HIS ABILITY TO QUICKLY AND ACCURATELY CARRY OUT HIS INTERNAL DEMONSTRATION OF CORRECTNESS, AND

  (2) TO PROVIDE TECHNIQUES TO MAKE EXPLICIT HIS REASONING SO IT CAN BE REVIEWED BY OTHERS

INSTRUCTOR NOTES

POINT OUT THE RANGE OF POSSIBILITIES FOR A DEMONSTRATION OF CORRECTNESS.

EMPHASIZE THE RANGE FROM ONE EXTREME -- A MATHEMATICAL PROOF -- TO THE OTHER EXTREME — THE PURE MENTAL EXERCISE.

GUIDE THINKING TOWARD SELECTION OF THE LAST POINT AS THE PRACTICAL MIDDLE GROUND.

EACH OF THESE THREE POINTS IS DISCUSSED MORE FULLY ON THE FOLLOWING THREE SLIDES.

DEMONSTRATION OF CORRECTNESS

- DEMONSTRATION OF CORRECTNESS CAN TAKE MANY FORMS
  - A FORMAL, MATHEMATICAL PROOF
  - A PURELY MENTAL EXERCISE IN THE PROGRAMMER'S HEAD
  - A SKETCH OF THE ESSENCE OF THE REASONING ATTACHED AS COMMENTS TO THE PROGRAM

INSTRUCTOR NOTES

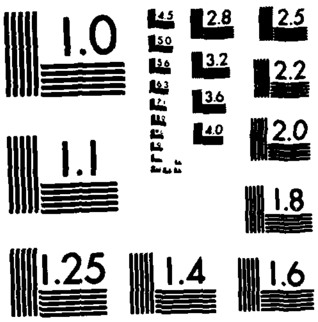
POINT OUT WHY FORMAL PROOF ISN'T PRACTICAL.

EMPHASIZE THIS SO THEY WILL REALIZE WE ARE INTERESTED IN REALLY PRACTICAL TECHNIQUES, NOT JUST ACADEMIC EXERCISES.

NOTE THE EXCEPTION OF SECURITY RELATED PROGRAMS. A NUMBER OF TOOLS HAVE BEEN DEVELOPED TO HELP IN THE TASK OF ACTUALLY PROVING PROGRAMS IN THIS FIELD. POINT OUT THAT THIS IS ENORMOUSLY EXPENSIVE AND IS A VERY SPECIAL UNDERTAKING THAT IS FAR REMOVED FROM THE REAL OF EVERYDAY PROGRAMMING.

AD-A143 501 ADA (TRADEMARK) TRAINING CURRICULUM PROGRAMMING METHODOLOGY M203 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA JUL 84 DAAB07-83-C-K514 6/8

UNCLASSIFIED F/G 5/9 NL


1.0
2.8
2.5
3.2
2.2
3.6
2.0
4.0
1.1
1.8
1.25
1.4
1.6

DEMONSTRATION OF CORRECTNESS

- FORMAL PROOF IS NOT SUITABLE FOR PRACTICAL PROGRAMMING
  - IT'S TOO COMPLICATED AND TIME CONSUMING
  - THE PROOF IS OFTEN AS COMPLICATED AS THE PROGRAM ITSELF

- A POSSIBLE EXCEPTION TO THIS IS IN THE AREA OF CRITICALLY IMPORTANT SECURITY KERNAL PROGRAMS, WHERE THE EXPENSE AND DIFFICULTY OF A PROOF MAY BE JUSTIFIED.

DEMONSTRATION OF CORRECTNESS

- THE PROBLEM WITH A PURE MENTAL EXERCISE AS A DEMONSTRATION OF CORRECTNESS IS THAT IT LEAVES NO TRACE EXCEPT IN THE PROGRAMMERS HEAD
- EVEN THAT TRACE GOES AWAY IF THE PROGRAMMER LEAVES THE PROJECT
- SUCH TRACES ALSO TEND TO ERODE WITH THE PASSAGE OF TIME

INSTRUCTOR NOTES

AFTER HAVING SHOT DOWN THE EXTREME POSITIONS, HERE IS THE REASONABLE, PRACTICAL MIDDLE GROUND.

EMPHASIZE THE WORD "CONSISTENTLY" IN THE SECOND BULLET. THERE IS GREAT VIRTUE IN RELIGIOUS CONSISTENCY IN FOLLOWING THE RULES. TRY TO EMPHASIZE THIS BY WORD (I.E. SAY IT EXPLICITLY) AND BY DEED (I.E. FOLLOW THE RULES IN ALL EXAMPLES AND EXERCISES).

ALSO EMPHASIZE THE WORD "COOPERATIVE" IN THE LAST POINT. THE IDEA OF REVIEW IS TO HELP THE AUTHOR, NOT JUDGE HIM.

DEMONSTRATION OF CORRECTNESS

- THE MOST PRACTICAL DEMONSTRATION OF CORRECTNESS IS COMMENTS IN THE PROGRAM SKETCHING THE ESSENTIAL POINTS IN THE CORRECTNESS ARGUMENT

- THERE ARE A FEW RULES THAT, IF FOLLOWED CONSISTENTLY, LEAD THE PROGRAMMER TO THINK THROUGH AND COMMENT HIS PROGRAMS IN A LUCID WAY

- THE COMMENTS PROVIDE THE BASIS FOR COOPERATIVE REVIEW OF THE PROGRAM

- THEY REMAIN IN THE PROGRAM AFTER THE PROGRAMMER IS GONE

INSTRUCTOR NOTES

ASSERTIONS MAY BE CLAIMS ABOUT THE VALUE OF A VARIABLE IN A PROGRAM AT A PARTICULAR POINT:

E.G. n>0

OR A CLAIM ABOUT THE RELATIONSHIPS AMONG VARIABLES:

E.G. m = abs(n) -- m is absolute value of n

OR A CLAIM ABOUT THE RELATIONSHIP BETWEEN PROGRAM VARIABLES AND THE OUTSIDE WORLD:

E.G. Max_T(i) is maximum temperature read from sensor i

MENTION THAT AN ASSERTION IS ALWAYS SOMETHING THAT CAN BE TRUE OR FALSE.

IN A CORRECT PROGRAM, ASSERTIONS WILL ALWAYS BE TRUE.

ASSERTIONS

- THE MOST IMPORTANT WEAPON IN THE CORRECTNESS DEMONSTRATION FIGHT IS THE <u>ASSERTION</u>

- AN ASSERTION IS SIMPLY A STATEMENT ABOUT WHAT RELATIONSHIPS HOLD AT SOME POINT IN A PROGRAM

- AN ASSERTION IS A CLAIM THAT A PARTICULAR STATEMENT IS TRUE AT SOME POINT IN A PROGRAM

INSTRUCTOR NOTES

READ THROUGH EACH OF THE EQUIVALENT WAYS OF EXPRESSING THIS ASSERTION.

THE ENGLISH AND MATHEMATICAL STATEMENTS ARE STRAIGHTFORWARD.

IN THE ADA FORM THE "ASSERT" PROCEDURE MAY BE THOUGHT OF AS A PROCEDURE WITH A SINGLE BOOLEAN ARGUMENT THAT RAISES SOME Assertion_Failure EXCEPTION IF ITS ARGUMENT IS FALSE. WHEN USED IN THIS WAY TO STATE AN ASSERTION, THE ADA CODE IS WRITTEN AS COMMENTS AND MAKES THE CLAIM THAT THE "ASSERT" STATEMENT WILL NEVER GET AN ARGUMENT THAT IS FALSE.

OFTEN AN ASSERTION EXPRESSED IN ADA WILL BE NOTHING MORE THAN A SINGLE BOOLEAN EXPRESSION.

ASSERTIONS

- AN ASSERTION MAY BE STATED IN VARIOUS WAYS:
  - ENGLISH -- ARRAY A IS SORTED IN ASCENDING ORDER
    -- THROUGH POSITION J
  - MATHEMATICS -- $A_i \leq A_{i+1}$ FOR $1 \leq i < J$
  - Ada

```
-- for i in 1 .. J-1
-- loop
--    Assert (A(i)<= A(i+1));
-- end loop;
```

- EACH OF THE ABOVE ASSERTIONS ARE ESSENTIALLY EQUIVALENT

INSTRUCTOR NOTES

IT SHOULD BE CLEAR HOW ASSERTIONS COULD BE EXPRESSED AS COMMENTS.

AS BEFORE, THE ASSERT PROCEDURE TAKES A BOOLEAN ARGUMENT AND RAISES AN Assertion_Failure EXCEPTION IF THE ARGUMENT IS FALSE. THE ASSERT PROCEDURE CALL COULD ACTUALLY BE CODED IN THE ADA PROGRAM. IT THUS PROVIDES NOT ONLY A CLAIM THAT THE ARGUMENT IS TRUE, BUT ALSO A DEFENSIVE PROGRAM CHECK AGAINST A LOGICAL ERROR.

THE IF STATEMENT FORM IS LESS OBVIOUSLY AN ASSERTION, BUT IT DOES CONVEY TO A READER THAT THE Boolean_Expression HAD BETTER BE TRUE FOR NORMAL PROGRAM EXECUTION TO PROCEED.

ASSERTIONS

- ASSERTIONS MAY BE PLACED IN AN Ada PROGRAM

  - AS COMMENTS

    ```
    -- ASSERTION STATEMENT
    ```

  - AS CALL ON AN Assert PROCEDURE

    ```
    Assert(Boolean_Expression);
    ```

  - AS IF STATEMENT

    ```
    if not Boolean_Expression then
       raise Error_Exception;
    end if;
    ```

INSTRUCTOR NOTES

THE GOAL OF THE CORRECTNESS DEMONSTRATION IS TO SHOW THAT, GIVEN THE ASSERTION AT THE ENTRY TO THE PROGRAM, WE CAN DEMONSTRATE THAT THE ASSERTION AT THE EXIT IS GUARANTEED TO BE TRUE.

ASSERTIONS

- THE DEMONSTRATION OF CORRECTNESS OF A PROGRAM STARTS WITH AN ASSERTION THAT STATES THE CONDITIONS ON ENTRY TO THE PROGRAM

AND ...

- ENDS WITH AN ASSERTION THAT DESCRIBES THE CONDITIONS AT THE END OF THE PROGRAM

- IN BETWEEN ARE OTHER ASSERTIONS AS NECESSARY TO ALLOW A READER OF THE PROGRAM TO CONVINCE HIMSELF THAT THE ASSERTIONS LOGICALLY FOLLOW FROM THE ONES ABOVE

- THE FOLLOWING SECTION DESCRIBES SOME RULES TO HELP DETERMINE HOW ASSERTIONS FOLLOW FROM PREVIOUS ASSERTIONS

INSTRUCTOR NOTES

THIS IS THE FIRST OF THE THEORY SLIDES. IT PROVIDES A VERY EASY INTRODUCTION TO THIS APPROACH.

THE EASIEST WAY TO EXPLAIN THIS IS TO POINT TO THE FLOWCHART AS YOU GO THROUGH THE STEPS OF THE ARGUMENT. IT IS ALMOST DEAD OBVIOUS FROM THE PICTURE.

NOTE THAT THERE IS AN EXAMPLE ON THE NEXT SLIDE, AND A FURTHER SLIDE ON PRACTICAL IMPLICATIONS.

SEQUENTIAL STATEMENTS

THEORY

IF

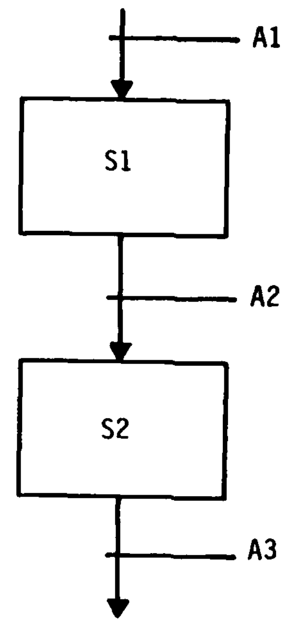
ASSERTION A1 TRUE BEFORE STATEMENT S1
LEAVES ASSERTION A2 TRUE AFTER S1

AND IF

ASSERTION A2 TRUE BEFORE STATEMENT S2
LEAVES ASSERTION A3 TRUE AFTER S2

THEN

ASSERTION A1 TRUE BEFORE THE SEQUENCE
S1; S2;
LEAVES ASSERTION A3 TRUE AFTER THE SEQUENCE



S1; S2;

INSTRUCTOR NOTES

POINT OUT HOW A2 FOLLOWS FROM A1 BY SAYING THAT THE NEW VALUE OF n (AFTER THE n := n+1; STATEMENT) IS EQUAL TO THE OLD VALUE MINUS ONE, AND THEREFORE THE PREVIOUS ASSERTION WHICH WAS TRUE OF n IS NOW TRUE OF n-1.

MORE FORMALLY THIS ARGUMENT CAN BE STATED AS SHOWN BELOW. THE FORM n(OLD) REFERS TO THE VALUE OF n BEFORE THE STATEMENT WHILE n(NEW) REFERS TO THE VALUE AFTER THE STATEMENT.

```
from A1: b = 2 ** n(old)
from statement n(new) = n(old) + 1 so n(old) = n(new)-1
therefore, substituting gives b = 2 ** (n(new)-1)
or, since the current value of n is n(new),
A2: b = 2 ** (n-1)
```

A3 FOLLOWS FROM A2 BY A SIMILAR ARGUMENT TO THAT SHOWN ABOVE.

THE IDEA IS TO MAKE THESE APPEAR TO BE OBVIOUS AND NOT GET BOGGED DOWN IN FORMALITY. BUT IF YOU GET PRESSED, YOU CAN DO THE MORE DETAILED ARGUMENT.

SEQUENTIAL STATEMENTS

EXAMPLE

```
--  Assert A1: b = 2 ** n

n := n + 1;

--  Assert A2: b = 2 ** (n-1)

b := 2 * b;

--  Assert A3: b = 2 ** n
```

INSTRUCTOR NOTES

IT SHOULD BE CLEAR THAT THIS CHAINING OF ASSERTIONS CAN BE EXTENDED TO A SEQUENCE OF MORE THAN TWO STATEMENTS.

THE LAST TWO POINTS ARE A PRACTICAL HINT. USUALLY A WHOLE SEQUENCE OF STATEMENTS IS CONSIDERED AS A SINGLE UNIT WHOSE EFFECT IS UNDERSTOOD AS A WHOLE. USUALLY IT ISN'T NECESSARY TO GO THROUGH THE WHOLE SEQUENCE IN DETAIL.

IN PRACTICAL TERMS, THIS MEANS THAT ALL THAT IS NECESSARY IS TO COMMENT ON THE PROGRAM WITH AN ASSERTION AT THE BEGINNING OF THE SEQUENCE AND AT THE END OF THE SEQUENCE.

SEQUENTIAL STATEMENTS

PRACTICAL CONSIDERATIONS

- THIS REASONING CAN BE EXTENDED TO ANY NUMBER OF STATEMENTS IN SEQUENCE
- IT IS RARELY NECESSARY TO FOLLOW ASSERTIONS IN DETAIL THROUGH A SEQUENCE OF STATEMENTS
- IN MOST CASES A SEQUENTIAL BLOCK OF STATEMENTS WILL BE CONSIDERED AS A SINGLE UNIT

INSTRUCTOR NOTES

THIS THEORY SLIDE IS A LITTLE MORE COMPLICATED THAN THE FIRST ONE, BUT REFERENCE TO THE FIGURE MAKES THE REASONING DEAD OBVIOUS.

FIRST EXPLAIN THE TWO PREMISES BY POINTING TO THE S1 AND S2 BOXES ON THE CHART.

THEN SHOW HOW THE IF STATEMENT ENSURES THAT A1 AND B WILL BE TRUE BEFORE S1 AND SIMILARLY HOW A1 AND not B WILL BE TRUE BEFORE S2.

FINALLY IT SHOULD BE CLEAR THAT NO MATTER WHICH WAY WE GO THROUGH THE STATEMENT, A2 WILL BE TRUE AT THE END.

CONDITIONAL STATEMENTS

THEORY

IF

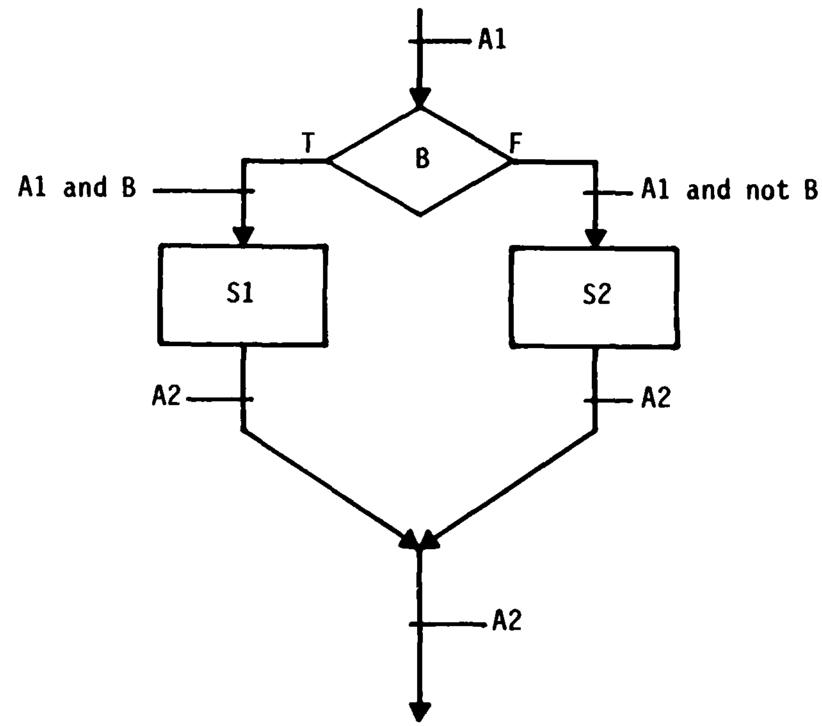
A1 TRUE AND CONDITION B TRUE BEFORE S1
LEAVES A2 TRUE AFTER S1

AND IF

A1 TRUE AND CONDITION B NOT TRUE BEFORE
S2 LEAVES A2 TRUE AFTER S2

THEN

A1 TRUE BEFORE THE CONDITIONAL
    if B then S1; else S2; end if;
LEAVES A2 TRUE AFTER THE CONDITIONAL



if B then S1; else S2; end if;

INSTRUCTOR NOTES

STEP THROUGH THIS EXAMPLE SHOWING HOW EACH BRANCH MAKES THE FINAL ASSERTION TRUE FOR ITS PART OF THE JOB.

THE FIRST ASSERTION COULD JUST AS WELL HAVE BEEN EMPTY (I.E. AN ASSERTION THAT IS ALWAYS TRUE). THE ASSERTION OF THE TYPE OF n IS MORE OF A PLACEHOLDER THAN AN IMPORTANT PART OF THE ARGUMENT.

YOU MIGHT POINT OUT THAT, IN PRACTICE, THE ASSERTION A2 WOULD BE WRITTEN ONLY AT THE END OF THE WHOLE IF STATEMENT RATHER THAN BEING REPEATED AFTER THE THEN AND ELSE STATEMENTS.

CONDITIONAL STATEMENTS

EXAMPLE

```
--  Assert A1: n is an integer

if n < 0
then                    --  Assert A1 and B: n is negative integer
     m := - n;
                        --  Assert A2: m is absolute value of n
else                    --  Assert A1 and not B: n is positive or zero integer
     m := n;
                        --  Assert A2: m is absolute value of n
end if;
--  Assert A2: m is absolute value of n
```

INSTRUCTOR NOTES

THE FIRST POINT IS IMPORTANT. THE PURPOSE OF THE WHOLE CONDITIONAL STATEMENT IS TO MAKE SOME OUTPUT ASSERTION TRUE. IT DOES THAT BY DIVIDING THE CIRCUMSTANCES INTO CASES CORRESPONDING TO THE CONDITIONAL BRANCHES AND MAKING THE OUTPUT ASSERTION TRUE FOR EACH BRANCH. IN GENERAL THE ACTIONS NEEDED TO MAKE THE OUTPUT ASSERTION TRUE ARE DIFFERENT FOR EACH BRANCH (THAT'S WHY THE CONDITIONAL IS THERE IN THE FIRST PLACE TO ALLOW SOMETHING DIFFERENT TO BE DONE IN DIFFERENT CASES).

HOW TO EXTEND THE REASONING TO THESE OTHER CASES IS THE SUBJECT OF THE EXERCISE ON THE NEXT SLIDE.

THE LAST POINT IS A PRACTICAL HINT TO LEAVE THEM WITH.

CONDITIONAL STATEMENTS

PRACTICAL CONSIDERATIONS

- A CONDITIONAL MUST MAKE SOME OUTPUT ASSERTION TRUE REGARDLESS OF THE BRANCH TAKEN THROUGH THE CONDITIONAL

- THE REASONING CAN BE EXTENDED TO OTHER TYPES OF CONDITIONALS

  INCLUDING if then end if

  if then elsif then ... else endif

  case statements

- THE MAIN POINT TO REMEMBER:

  CHECK ALL BRANCHES OF A CONDITIONAL TO ENSURE EACH ONE ACCOMPLISHES THE PURPOSE OF THE CONDITIONAL IN THE CIRCUMSTANCES SELECTED BY THE BRANCH CONDITION

INSTRUCTOR NOTES

YOU MIGHT TELL THEM FIRST TO DRAW UP THE FLOWCHART CORRESPONDING TO THE DIFFERENT CASES AND THEN ANNOTATE THE FLOWCHART WITH THE ASSERTIONS THAT CAN BE MADE AT THE VARIOUS POINTS. IF NECESSARY DRAW THE FLOWCHARTS FOR THEM AND LET THEM WORK FROM THERE.

THE NEXT PAGE PROVIDES SPACE TO WORK OUT THE EXERCISE SOLUTION.

CONDITIONAL STATEMENTS

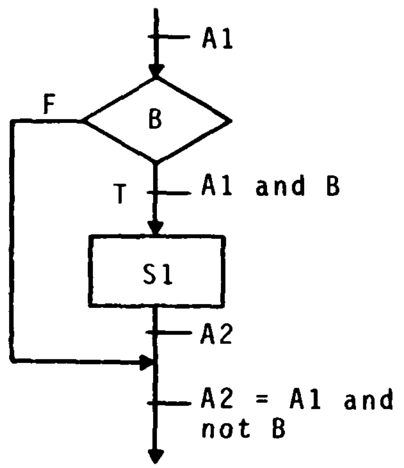
EXERCISE

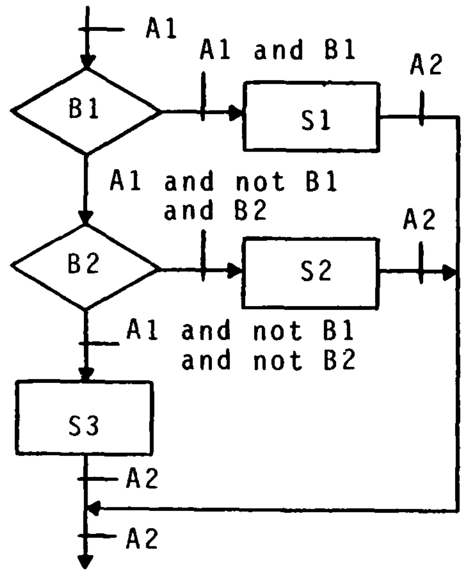
- WHAT IS THE PATTERN OF ASSERTIONS FOR THE FOLLOWING FORMS OF CONDITIONALS:

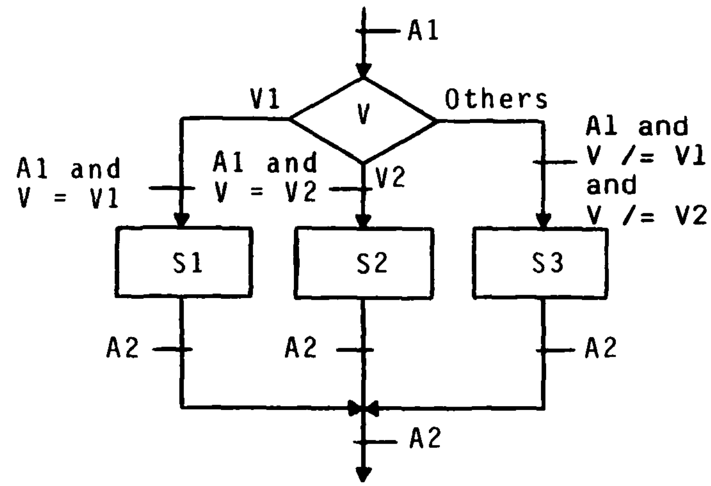
```
if B then S1; end if;

if B1 then S1; elsif B2 then S2; else S3 end if;

case V is when V1 => S1; when V2 => S2; when others => S3; end case;
```








```
if B then
   S1;
end if;
```

```
if B1 then
   S1;
elsif B2 then
   S2;
end if;
```

```
case V is
   when V1     => S1;
   when V2     => S2;
   when others => S3;
end case;
```

CONDITIONAL STATEMENTS

EXERCISE SOLUTION

INSTRUCTOR NOTES

AGAIN THE REASONING IS MOST CLEARLY SHOWN BY REFERENCE TO THE FLOWCHART.

POINT OUT IN PASSING HOW ASSERTION A1 IS TRUE EACH TIME AROUND THE LOOP. THIS WILL BE DISCUSSED IN MORE DETAIL LATER WHEN WE TALK ABOUT LOOP INVARIANTS.

ITERATIVE STATEMENTS

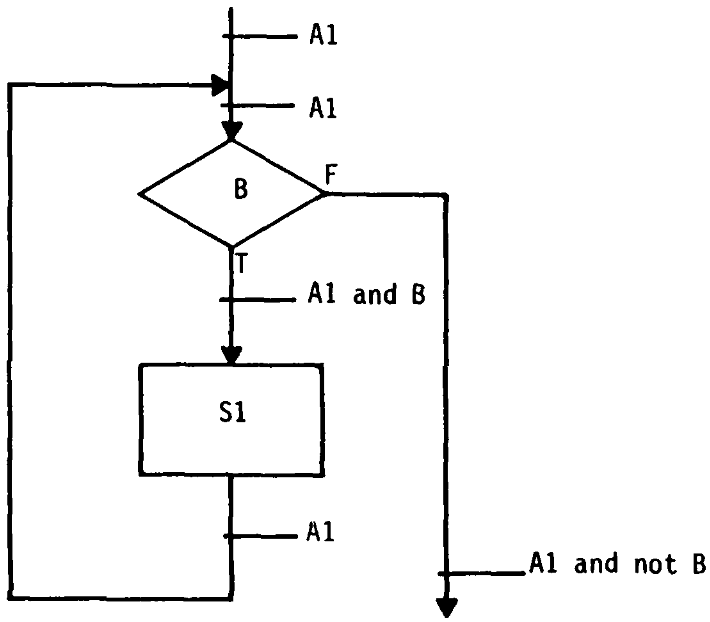
THEORY

IF

A1 TRUE AND CONDITION B TRUE BEFORE S1
LEAVES A1 TRUE AFTER S1

THEN

A1 TRUE BEFORE THE LOOP:
while B loop S1; end loop;
LEAVES A1 TRUE AND B NOT TRUE AFTER
THE LOOP



while B loop S1; end loop;

INSTRUCTOR NOTES

THIS IS SIMILAR TO THE PRECEDING LOOP EXCEPT IT CONTAINS AN EXIT IN THE MIDDLE OF THE LOOP. AS YOU EXPLAIN THIS POINT OUT THAT S1 REPRESENTS ALL THE STATEMENTS BEFORE THE EXIT STATEMENT AND THAT S2 REPRESENTS ALL THE STATEMENTS AFTER THE EXIT.

NOTE AGAIN HOW A1 IS TRUE BEFORE AND AFTER EACH PASS THROUGH THE LOOP.

ITERATIVE STATEMENTS

THEORY

IF

A1 TRUE BEFORE S1

LEAVES A2 TRUE AFTER S1

AND IF

A2 TRUE AND B NOT TRUE BEFORE S2

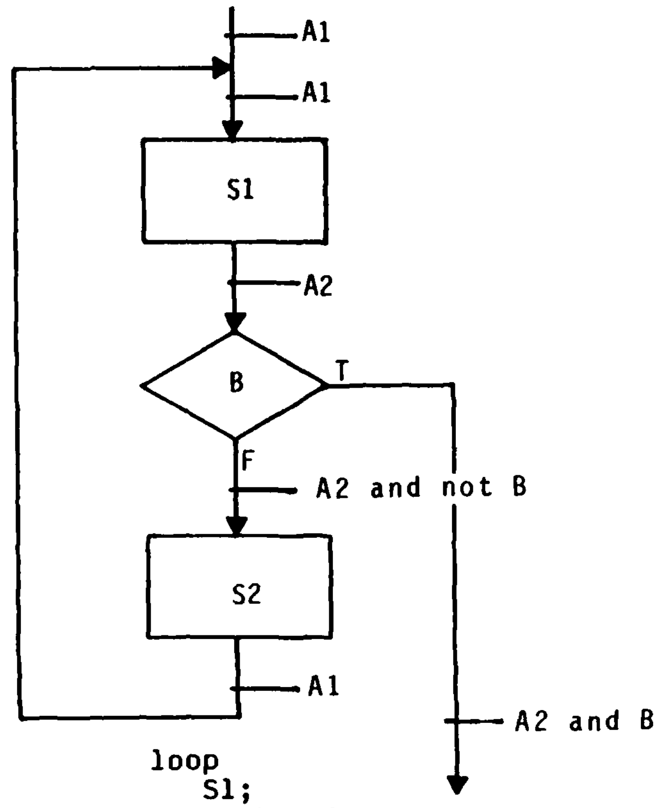
LEAVES A1 TRUE AFTER S2

THEN

A1 TRUE BEFORE THE LOOP:

loop S1; exit when B; S2; end loop;

LEAVES A2 TRUE AND B TRUE AFTER THE LOOP




```
loop
   S1;
exit when B;
   S2;
end loop;
```

INSTRUCTOR NOTES

ASSERTION A1 IS THE LOOP INVARIANT -- WE'VE LOOKED IN A UP TO BUT NOT INCLUDING POSITION J AND HAVEN'T FOUND THE VALUE Val. IT IS TRIVIALLY TRUE ON ENTRY TO THE LOOP SINCE THERE'S NOTHING IN A BELOW POSITION 1.

AT THE TOP OF THE LOOP BODY A1 IS TRUE AND SO IS THE WHILE CONDITION. THIS MEANS THAT J HASN'T RUN PAST THE END AND THAT Val ISN'T IN POSITION J (OR ELSE THE WHILE CONDITION WOULD BE FALSE). SINCE THE VALUE ISN'T BEFORE POSITION J (FROM A1) AND IT ISN'T IN POSITION J (FROM WHILE) THEN IT ISN'T THERE THROUGH POSITION J.

AFTER THE INCREMENT OF J, THE INVARIANT IS RESTORED -- THE VALUE Val WASN'T IN A THROUGH THE OLD VALUE OF J SO IT'S NOT THERE UP TO BUT NOT INCLUDING THE NEW VALUE OF J.

AT THE END OF THE LOOP, THE WHILE CONDITION IS FALSE. THUS EITHER J N (THE NOT FOUND CONDITION) OR Val IS IN POSITION J (THE FOUND CASE). IN EITHER CASE, THE INVARIANT ASSURES THAT Val IS NOT BEFORE J. THIS MEANS THAT IN THE NOT FOUND CASE, IT'S NOT IN THE ARRAY AT ALL, AND IN THE FOUND CASE THAT WHAT WE'VE FOUND IS THE FIRST OCCURRENCE OF Val.

ITERATIVE STATEMENTS

EXAMPLE

```
J := 1;
                                            -- A1: Val not in A before
                                            --     position J
while J <= N and then A(J) /= Val
loop
                                            -- A1 and B: Val not in A
                                                           including position J

   J := J + 1:
end loop;
                                            -- A1 and not B:
                                            --    J > N or Val is in A
                                            --    at position J
                                            --    (and not before)
```

INSTRUCTOR NOTES

RECALL THAT FOR EACH FORM OF LOOP THERE WAS AN ASSERTION THAT IS TRUE AT THE BEGINNING AND THE END OF THE LOOP BODY.

TO DETERMINE WHAT THE LOOP INVARIANT IS IT IS HELPFUL TO DRAW A PICTURE OF THE SITUATION AT SOME INTERMEDIATE ITERATION OF THE LOOP. THE LOOP INVARIANT IS THE STATEMENT OF THE IMPORTANT RELATIONSHIPS IN THE PICTURE.

THE PARENTHESIZED COMMENT ABOUT A DERIVATIVE OF THE LOOP INVARIANT REFERS TO THE SITUATION WHERE THERE IS AN EXIT FROM THE MIDDLE OF THE LOOP. THE LOOP INVARIANT IS TRUE ONLY AT THE TOP OR BOTTOM OF THE LOOP. SOME OTHER ASSERTION (A2 IN THE FLOWCHART) MAY BE TRUE AT AN EXIT POINT IN THE MIDDLE OF THE LOOP.

THE COMMENTING SUGGESTION IS ANOTHER PRACTICAL HINT.

ITERATIVE STATEMENTS

PRACTICAL CONSIDERATIONS

- NOTE THAT FOR ALL LOOPS THERE IS AN ASSERTION THAT IS TRUE EACH TIME AROUND THE LOOP

  THIS ASSERTION IS CALLED THE LOOP INVARIANT

- THE LOOP INVARIANT DESCRIBES A SNAPSHOT OF WHAT THE SITUATION IS EACH TIME THROUGH THE LOOP

- AT THE END OF THE LOOP YOU KNOW THAT THE LOOP INVARIANT (OR SOME DERIVATIVE OF IT) IS STILL TRUE PLUS YOU KNOW THAT THE TERMINATION CONDITION IS MET

- ALMOST ALL LOOPS SHOULD BE COMMENTED WITH THE LOOP INVARIANT ASSERTION

INSTRUCTOR NOTES

IN THE CASE OF MULTIPLE EXITS, THE CONDITIONS THAT HOLD AT THE END OF THE LOOP IS JUST THE LOGICAL "OR" OF THE EXIT CONDITIONS DUE TO EACH EXIT.

THE CONDITIONS DUE TO EACH EXIT ARE JUST THE CONDITIONS THAT EXIT AT THE POINT OF THE EXIT COUPLED WITH THE BOOLEAN CONDITION FOR MAKING THE EXIT.

THE LAST POINT IS A PRACTICAL HINT. AS YOU ARE REVIEWING A PROGRAM GO THROUGH THE MENTAL EXERCISE OF ASKING WHAT IS TRUE FOR EACH POSSIBLE EXIT FROM EACH LOOP.

ITERATION STATEMENTS

PRACTICAL CONSIDERATIONS

- THE GENERAL PHILOSOPHY OF CORRECTNESS DEMONSTRATION CAN BE EXTENDED TO LOOPS WITH MULTIPLE EXITS.

  WHEN THE EXIT IS NOT AT THE TOP OR BOTTOM OF THE LOOP, IT IS NOT THE LOOP INVARIANT ASSERTION THAT IS TRUE, BUT SOME DERIVATIVE OF IT THAT IS TRUE AT THE POINT OF THE EXIT

- THE ASSERTIONS THAT HOLD AT THE EXIT POINT OF A LOOP DEPEND ON THE ASSERTIONS THAT HOLD AT THE POINT OF THE LOOP EXIT TOGETHER WITH THE TERMINATION CONDITION

- EACH LOOP EXIT SHOULD BE CHECKED TO ENSURE THAT THE ASSERTION AT THE POINT OF THE EXIT COMBINED WITH THE CONDITION OF THE EXIT RESULT IN THE PROPER CONDITIONS AT THE END OF THE LOOP

INSTRUCTOR NOTES

THIS IS JUST A "THINKING SHORTCUT" TO SAVE TIME AND IMPROVE ACCURACY OF THINKING OUT THE EFFECTS OF ELSE CLAUSES OR OF LEAVING WHILE LOOPS.

SLOWLY VERBALIZING THE FORMULAS AND THEIR INVERSES WITH PAUSES TO INDICATE GROUPING MAKES THE CONCLUSIONS SEEM OBVIOUS.

"BAG OF TRICKS"

- IN DOING CORRECTNESS ARGUMENTS, IT IS OFTEN NECESSARY TO DETERMINE THE INVERSE (NEGATION) OF A GIVEN CONDITION

- THE SIMPLE TRICK KNOWN AS <u>DEMORGAN'S LAWS</u> HELPS DO THAT EASILY FOR COMPLICATED ASSERTIONS:

    TO INVERT A CONDITION, INVERT EACH TERM AND CHANGE "and" TO "or" AND VICE VERSA

ASSERTION	INVERSE
A1 and A2	not A1 or not A2
A1 or A2	not A1 and not A2
i<n and A(i)/=V	i>=n or A(i)=V
end_line or end_page	not end_line and not end_page

INSTRUCTOR NOTES

THIS SECTION DESCRIBES SOME MANAGEMENT TECHNIQUES THAT CAN BE USED TO IMPROVE THE QUALITY OF LARGE SOFTWARE SYSTEMS.

ENSURING RELIABILITY

PROJECT MANAGEMENT TECHNIQUES

INSTRUCTOR NOTES

THIS LIST IS NOT EXHAUSTIVE OF ALL MANAGEMENT TECHNIQUES APPLICABLE TO THE CODING PHASE.

THESE ARE A REPRESENTATIVE SET OF TECHNIQUES THAT DO CONTRIBUTE TO RELIABILITY.

PROJECT MANAGEMENT TECHNIQUES

- THERE ARE A NUMBER OF MANAGEMENT PROCEDURES AND TECHNIQUES THAT HAVE EVOLVED TO HELP IMPROVE CODE RELIABILITY
  - CODE READING
  - EGOLESS PROGRAMMING
  - UNIT DEVELOPMENT FOLDERS

INSTRUCTOR NOTES

COMMENTING SHOULD BE IN WRITING DIRECTLY ON A COPY OF THE PROGRAM.

CODE READING

METHOD

- IN ADDITION TO THE PRIMARY AUTHOR, A DESIGNATED CODE READER IS ASSIGNED TO EACH MODULE
- THE READER READS AND COMMENTS TO THE AUTHOR ON THE MODULE
- THE READER IS EQUALLY AS RESPONSIBLE FOR THE CODE RELIABILITY AS THE AUTHOR
- THE READER IS NORMALLY A PEER OF THE AUTHOR WITH HIS OWN SET OF MODULES TO DEVELOP AS AUTHOR

INSTRUCTOR NOTES

THERE IS SOME DISAGREEMENT ABOUT WHETHER OR NOT THE FIRST CODE READING SHOULD BE BEFORE OR AFTER THE INITIAL COMPILATION HAS BEEN DONE. DOING IT BEFORE MAY CATCH SOME SYNTAX ERRORS BUT WILL TAKE MORE OF THE READER'S TIME LOOKING FOR SUCH ERRORS. DOING IT AFTER THE FIRST COMPILE ALLOWS THE READER TO CONCENTRATE ON THE PROGRAM SEMANTICS.

IN ANY CASE READING SHOULD ALWAYS BE DONE ON A PRINT OUT OF THE MACHINE READABLE PROGRAM RATHER THAN ON THE AUTHOR'S HANDWRITTEN ORIGINAL.

EMPHASIZE ON THE LAST POINT THAT WE MUST COMMENT ON WHAT THE AUTHOR WROTE, NOT WHAT HE INTENDED TO WRITE.

CODE READING

METHOD

- INITIAL READING SHOULD BE DONE SHORTLY AFTER A MACHINE READABLE VERSION OF THE MODULE IS AVAILABLE

- INITIAL READING CAN BE EITHER BEFORE OR AFTER INITIAL COMPILATION AND SYNTAX ERROR REMOVAL
  - IF DONE BEFORE, SOME SYNTAX ERRORS CAN BE CAUGHT BUT THE EXTRA READER EFFORT MAY NOT BE WORTH WHILE

- FINAL READING SHOULD BE WHEN THE CODING AND UNIT TESTING IS ALMOST FINISHED

- THE READER SHOULD BASE COMMENTS STRICTLY ON THE CODE AND ITS WRITTEN DOCUMENTATION -- NOT ON VERBAL DISCUSSION WITH THE AUTHOR

INSTRUCTOR NOTES

THE "MANAGEMENT EFFORT" REFERRED TO IN THE LAST BULLET IS SOME TYPE OF CHECKUP ON WHETHER THE READING IS BEING DONE COUPLED WITH SOME FORM OF ENCOURAGEMENT TO GET IT DONE.

IT HAS BEEN FOUND ON MANY PROJECTS THAT START WITH THE BEST OF INTENTIONS, THAT READING IS OFTEN DISPENSED WITH UNILATERALLY BY THE READERS AS THEY GET PRESSURED TO COMPLETE THEIR OWN AUTHORING ASSIGNMENTS.

THE BEST INTERESTS OF THE *PROJECT ARE SERVED WHEN THE READING IS FOLLOWED THROUGH.*

CODE READING

METHOD

- CODE READING IS VERY EFFECTIVE WHEN SUFFICIENT TIME IS ALLOCATED TO IT
- THE ASSIGNED READER NEEDS ABOUT 20% OF THE ASSIGNED CODING TIME TO DO AN EFFECTIVE JOB OF READING
- EXPERIENCE SHOWS THAT MANAGEMENT EFFORT NEEDS TO BE APPLIED TO ENSURE THE CODE READING IS KEPT UP

INSTRUCTOR NOTES

IF THE READER CAN UNDERSTAND THE MODULE, THEN WE KNOW THAT IT IS UNDERSTANDABLE TO AT LEAST ONE PERSON BESIDES ITS AUTHOR.

THE LAST POINT IS ESPECIALLY INTERESTING TO PROJECT MANAGERS.

CODE READING

ADVANTAGES

- PROVIDES AN "ACID TEST" FOR MODULE UNDERSTANDABILITY
- ENSURES THE KEYS REQUIRED FOR UNDERSTANDABILITY ARE INCLUDED WITHIN THE MODULE AND ITS DOCUMENTATION AND NOT JUST IN THE AUTHOR'S HEAD
- DETECTS BUGS IN THE MODULE AND ITS INTERFACES
- PROVIDES BROADER KNOWLEDGE AMONG PROJECT STAFF ABOUT DETAILS OF THE PROGRAM

INSTRUCTOR NOTES

EGOLESS PROGRAMMING IS CLEARLY A GOOD IDEA.

ITS NOT SO CLEAR TO SEE HOW TO IMPLEMENT IT IN SOME ORGANIZATIONS.

IF THERE IS TIME, ENCOURAGE A DISCUSSION OF THIS TOPIC IN THE CLASS.

EGOLESS PROGRAMMING

- COOPERATIVE CODING TECHNIQUES SUCH AS CODE READING WORK FOR BETTER IN AN EGOLESS ENVIRONMENT

  I.E. ONE IN WHICH EACH PERSON'S SELF IMAGE (EGO) IS <u>NOT</u> BOUND UP WITH HIS OR HER PROGRAM

- THIS IMPLIES THAT ALL MEMBERS OF THE PROGRAMMING GROUP ARE WILLING TO ACCEPT CONSTRUCTIVE CRITICISM WITHOUT CONSIDERING IT TO BE PERSONAL CRITICISM

- MANAGEMENT MUST TAKE THE LEAD IN CREATING THIS ENVIRONMENT BY JUDGING PROGRAMMERS ONLY ON THEIR <u>RESULTS</u> NOT ON THE NUMBER OF PROBLEMS THAT COME UP IN ACHIEVING THE RESULTS

- ACTIVE COOPERATION AMONG GROUP MEMBERS IS REQUIRED TO MAINTAIN THIS TYPE OF ENVIRONMENT.

INSTRUCTOR NOTES

*THIS IS A CORRECT SOLUTION. NOTE ERRORS MARKED WITH -- **. SHOW HOW THESE ERRORS COULD*

BE FOUND USING OUR RULES.

```
I := 0;                          -- ** 1 should be 0
                                    (a section below not satisfied for initial use)
                                 -- A is sorted in ascending order
                                 -- from position N-I+1 to position N
while I < N
loop
   J := 1;
                                 -- A(J) is max value in positions 1 through J
   while J<N-I
   loop
      if A(J) > A(J+1)
      then interchange (A(J), A(J+1));
      end if;
                                 -- A(J)  = A(J+1)
   J := J+1;                     -- ** missing increment statement (loop invariant
   end loop;                     -- for inner loop doesn't follow)
                                 -- A (N-I) is max value in positions 1 thourgh N-I
I := I+1;
end loop;
                                 -- A is sorted in ascending order
                                 -- from position 1 to position N
```

CODE READING

EXERCISE

```
I := 1;                                    -- A is sorted in ascending order
                                              from position N-I+1 to N
while I< N
loop

   J := 1;
                                           -- A(J) is max value in positions 1 through J
   while J<N-I
   loop
      if A(J) > A(J+1) then
      Interchange (A(J), A(J+1));
      end if;
                                           -- A(J) <= A(J+1)
   end loop;
                                           -- A(N-I) is max value in position 1 through N-I
I := I+1;
end loop;
                                           -- A is sorted in ascending order
                                              from position 1 to position N
```

INSTRUCTOR NOTES

THE NEXT SLIDE SHOWS AN EXAMPLE OF A UDF COVER THAT CONTAINS FIELDS FOR RECORDING THE STATUS INFORMATION.

THE FOLDER IS A REAL FOLDER, OPEN ON ONLY ONE SIDE, THAT CAN BE USED TO PHYSICALLY HOLD LISTINGS, DOCUMENTS, ETC.

IT SHOULD BE CLEAR THAT THE PHYSICAL FOLDERS COULD BE REPLACED BY AN AUTOMATED EQUIVALENT. THIS WILL PROBABLY HAPPEN IN THE NEXT TWO YEARS.

UNIT DEVELOPMENT FOLDERS

- A UNIT DEVELOPMENT FOLDER (UDF) PROVIDES A SINGLE PLACE FOR KEEPING INFORMATION ABOUT A SINGLE MODULE
  - INCLUDING STATUS INFORMATION (WHEN CODED, READ, COMPILED, TESTED, INTEGRATED, ETC.)
  - AND ACTUAL WORK PRODUCT
    - CODE LISTINGS
    - MODULE DOCUMENTATION
    - UNIT TEST PLAN
    - UNIT TEST RESULTS
    - BUG REPORTS AND FIXES
- CURRENTLY PHYSICAL FOLDERS ARE USED. SHORTLY THIS WILL BE REPLACED BY COMPUTER FILES CONTAINING THE SAME COLLECTED INFORMATION.

INSTRUCTOR NOTES

THIS FORM IS PRINTED ON THE OUTSIDE OF THE FOLDER.

IT IS FILLED IN TO GIVE THE CURRENT STATUS OF THE MODULE.

UNIT DEVELOPMENT FOLDERS

UNIT DEVELOPMENT FOLDER COVER SHEET

MODULE TITLE ______________________ CUSTODIAN ______________

DESCRIPTION ____________________________

MNEMONIC IDENTITY __________ UNIQUENESS PREFIX ____ INITIATION DATE ______

REVISION NUMBER __________

SECTION NUMBER	DESCRIPTION	DATE STARTED	DATE COMPLETED	ASSIGNED TO	CP APPROVAL & DATE	QA APPROVAL & DATE	DELIVERABLE FORM*
1	Requirements Definition						
2	Global Environment						
3	PDL						
4	Internal Design Review						
5	Code						
6	Code Reading						
7	Unit Test Scenarios						
8	Test Results						
9	CP Integration Testing						
10	System Build Integration Testing						
11	Placement under Configuration Management						
12	Problem Reports						

TERMINATION DATE __________ Approval ______________

REPLACEMENT MODULE NAMES ____________________

INSTRUCTOR NOTES

ENSURING RELIABILITY

MODULE DOCUMENTATION

INSTRUCTOR NOTES

THE LAST POINT HERE IS THE MOST IMPORTANT.

IT IS IMPORTANT TO EMPHASIZE HERE THAT THE ACTUAL SOURCE CODE IS A VERY IMPORTANT PART OF THE MODULE DOCUMENTATION. THIS IS EXPANDED ON IN THE NEXT SLIDE.

MODULE DOCUMENTATION

- MOST PROGRAMS (ESPECIALLY FOR EMBEDDED SYSTEMS) HAVE A LIFETIME LONGER THAN THEIR AUTHORS ASSIGNMENT TO THE PROJECT

  EVEN IF THE AUTHOR STAYS WITH A PROJECT INTO THE MAINTENANCE PHASE, HE WILL FORGET THE DETAILS OF INDIVIDUAL MODULES

- MAINTENANCE OF EMBEDDED PROGRAMS IS OFTEN THE RESPONSIBILITY OF A COMPLETELY DIFFERENT ORGANIZATION THAN THE DEVELOPING ORGANIZATION

- THE MODULE DOCUMENTATION (INCLUDING THE SOURCE CODE ITSELF) MUST CONTAIN SUFFICIENT INFORMATION FOR THE MODULE TO REMAIN UNDERSTANDABLE TO A NEW READER THROUGHOUT ITS ENTIRE LIFESPAN

INSTRUCTOR NOTES

AS IS STATED IN THE LAST POINT HERE, ADA WAS DESIGNED TO ALLOW PROGRAMS TO BE WRITTEN IN A READABLE MANNER. THIS WAS DONE PRECISELY BECAUSE EXTERNAL DOCUMENTATION TENDS TO GET OUT-OF-DATE AND BECOME UNRELIABLE.

MODULE DOCUMENTATION

ROLE OF THE SOURCE PROGRAM

- THE ULTIMATE DOCUMENTATION OF A MODULE IS THE SOURCE CODE ITSELF
- IT IS THE ONE PIECE OF DOCUMENTATION GUARANTEED TO BE UP-TO-DATE
- EXTERNAL DOCUMENTATION DESPITE THE BEST LAID PLANS (CONFIGURATION CONTROL AND OTHERWISE) OFTEN GOES ASTRAY
- CONSISTENTLY FOLLOWING TECHNIQUES SUCH AS THE ONES DESCRIBED IN THIS COURSE WILL GO A LONG WAY TOWARD PRODUCING CLEAR, UNDERSTANDABLE PROGRAMS THAT CAN STAND UP TO YEARS OF MAINTENANCE
- THE FEATURES OF Ada WERE DESIGNED TO ENCOURAGE READABILITY OVER WRITABILITY FOR EXACTLY THIS READABILITY

INSTRUCTOR NOTES

THERE IS NOT AN EASY SOLUTION TO THE PROBLEM POINTED OUT HERE OF ENSURING THE AVAILABILITY OF ACCURATE DESIGN LEVEL DOCUMENTATION. THE USE OF A PDL TO EXPRESS THE DESIGN MAY HELP, BUT WON'T SOLVE THE PROBLEM.

MODULE DOCUMENTATION

HIGHER LEVEL DOCUMENTATION

- THE ESSENTIAL INFORMATION MISSING FROM ANY FORM OF MODULE LEVEL DOCUMENTATION IS THE HIGHER LEVEL OVERVIEW OF HOW THE INDIVIDUAL MODULES FIT TOGETHER INTO THE WHOLE PROGRAM

- IDEALLY THIS INFORMATION SHOULD BE SUPPLIED IN A DESIGN SPECIFICATION (B SPEC), HOWEVER SUCH DOCUMENTS ARE OFTEN NOT UPDATED TO REFLECT CHANGES DURING IMPLEMENTATION

- PROGRAM MANAGERS ON BOTH THE DEVELOPER AND CUSTOMER SIDE MUST ENSURE THAT THIS LEVEL OF DOCUMENTATION IS AVAILABLE IN USABLE FORM SINCE IT IS ESSENTIAL TO THE CONTINUED MAINTENANCE OF THE PROGRAM

INSTRUCTOR NOTES

MIL-STD-48. AND 490 ARE MOST COMMONLY USED BY THE ARMY AND THE AIR FORCE. MIL-STD-1679 IS A NAVY STANDARD. MIL-STD-SDS IS A NEW STANDARD (THAT IS NOT YET APPROVED) ESSENTIALLY INCORPORATING THE 1679 APPROACH TO SOFTWARE DEVELOPMENT.

THERE IS NOT USUALLY A CHOICE (ON THE CONTRACTOR SIDE) OF THE MODULE DOCUMENTATION STANDARDS TO BE USED.

AS USE OF ADA INCREASES, IT MAY BE POSSIBLE TO REPLACE SOME OF THE EXTERNAL DOCUMENTS CURRENTLY REQUIRED BY CODING STANDARDS THAT ENSURE THAT THE ESSENTIAL INFORMATION WILL GET INCORPORATED INTO THE SOURCE CODE.

MODULE DOCUMENTATION STANDARDS

- THERE ARE SEVERAL DIFFERENT MODULE DOCUMENTATION STANDARDS IN USE TODAY:

C SPECIFICATION	MIL-STD-483 & -490
PROGRAM DESIGN SPECIFICATION	MIL-STD-1679
SOFTWARE DETAILED DESIGN	MIL-STD-SDS & R-DID-111

- THE EXACT MODULE DOCUMENTATION STANDARD TO BE FOLLOWED IS USUALLY SPECIFIED BY THE CUSTOMER

INSTRUCTOR NOTES

THESE ARE EXAMPLES OF DOCUMENTATION SUPPORT TOOLS THAT ARE CURRENTLY IN USE.

MODULE DOCUMENTATION TOOLS

- IT IS POSSIBLE TO BUILD TOOLS THAT HELP IN THE PRODUCTION AND MAINTENANCE OF DETAILED MODULE DOCUMENTS SUCH AS THESE USED BY THE ALS PROJECT:

GENSKEL — GENERATE SKELETON

PRODUCES A SKELETON OF A STANDARD FORM Ada PROGRAM FROM THE C-SPECIFICATION, INCLUDING HEADER COMMENTS, SUBPROGRAM HEADING WITH ARGUMENT DECLARATIONS, AND SUBPROGRAM ENDING

LEKSNEG — INVERSE OF GENSKEL (GENSKEL SPELLED BACKWARDS)

PRODUCES A C-SPECIFICATION WRITEUP FROM A STANDARD FORM Ada PROGRAM

GENSKEL IS USED DURING INITIAL DEVELOPMENT WHILE LEKSNEG IS USED TO REDO THE DOCUMENTATION WHEN CHANGES OCCUR

INSTRUCTOR NOTES

ENSURING RELIABILITY

UNIT TESTING

INSTRUCTOR NOTES

WITH THIS SLIDE, POINT OUT THE DIFFERENT TYPES OF TESTS AND WHAT THE PURPOSE OF EACH IS.

STATE THAT WE ARE CONCENTRATING ON UNIT TEST IN THIS COURSE.

UNIT TESTING

CLASSES OF TESTS

- UNIT TEST

  CHECKS THE CORRECT FUNCTIONING OF A SINGLE MODULE

- INTEGRATION TEST

  CHECKS THE CORRECT FUNCTIONING OF A COLLECTION OF MODULES

- ACCEPTANCE TEST

  CHECKS THE CORRECT FUNCTIONING OF A WHOLE SYSTEM

INSTRUCTOR NOTES

EMPHASIZE THE NEED FOR A WRITTEN UNIT TEST PLAN, BUT POINT OUT THAT THE PLAN CAN BE WRITTEN IN AN INFORMAL WAY.

TEST SCAFFOLDING (STUBS, DRIVERS, TEST DATA FILES) ARE OFTEN THROWN AWAY (OR JUST LOST TRACK OF) WHEN THE TESTS ARE FIRST PASSED. IT IS WORTHWHILE TO KEEP THIS MATERIAL IN AN ORGANIZED FORM SO THE UNITS CAN BE RETESTED IF CHANGES ARE MADE OR IF "FUNNY" PROBLEMS SHOW UP.

UNIT TESTING

PROCEDURES

- THERE SHOULD BE A WRITTEN UNIT TEST PLAN FOR EACH UNIT
- THE UNIT TEST PLAN IS AN <u>INFORMAL</u> DOCUMENT LISTING THE TESTS TO BE RUN, WHAT THEY TEST, AND THE CRITERIA FOR PASSING
- THE UNIT TEST PLAN AND THE CORRESPONDING TEST RESULTS SHOULD BE STORED IN THE UNIT DEVELOPMENT FOLDER
- THE READER ASSIGNED TO THE UNIT SHOULD ALSO REVIEW AND COMMENT ON THE UNIT TEST PLAN
- THE UNIT TEST PLANS AND RESULTS ARE SUBJECT TO REVIEW BY THE QA ORGANIZATION
- UNIT TEST FILES INCLUDING STUBS, DRIVERS, DATA FILES, ETC. SHOULD BE SAVED ON LINE SO THEY CAN BE REPEATED IF THE UNIT CHANGES

INSTRUCTOR NOTES

ALTHOUGH UNIT TEST IS THE MODULE PROGRAMMER'S RESPONSIBILITY, THE READER AND THE QA ORGANIZATION MAY CONTRIBUTE.

THE LIST OF UNIT TEST CONTENTS IS TAKEN FORM MIL-STD-SDS.

UNIT TESTING

- UNIT TEST IS THE RESPONSIBILITY OF THE MODULE PROGRAMMER
- THE UNIT TEST SHOULD SHOW:
  - CORRECTNESS OF COMPUTATIONS USING NOMINAL, SINGULAR, AND EXTREME DATA VALUES
  - CORRECT OPERATION FOR VALID AND INVALID DATA INPUT
  - CORRECT HANDLING OF ALL DATA OUTPUTS INCLUDING ERROR AND INFORMATION MESSAGES
  - THAT ALL EXECUTABLE STATEMENTS OPERATE CORRECTLY
  - THAT ALL BRANCHES OPERATE CORRECTLY

INSTRUCTOR NOTES

EVEN THOUGH WE HAVE GONE THROUGH A DEMONSTRATION OF CORRECTNESS, A THOROUGH TEST PROGRAM IS REQUIRED FOR EMBEDDED SYSTEMS SOFTWARE BECAUSE WE OCCASIONALLY MAKE MISTAKES IN DOING SUCH DEMONSTRATIONS.

UNIT TESTING

- RECALL THAT A PROGRAMMER HAS AN OBLIGATION TO DO ALL HE CAN TO ENSURE THAT HIS PROGRAMS ARE CORRECT

- A DEMONSTRATION OF CORRECTNESS GOES A LONG WAY TOWARD THIS GOAL

- ANY DEMONSTRATION OF CORRECTNESS MUST BE BACKED UP WITH A THOROUGH TESTING PROGRAM
  - WOULD YOU WANT TO RIDE IN A PLANE WHICH HAD ONLY BEEN DEMONSTRATED SAFE ON PAPER?

INSTRUCTOR NOTES

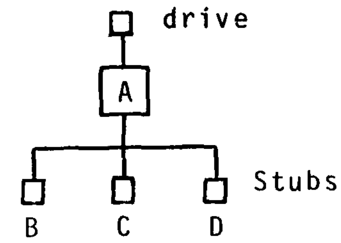
THERE ARE CHOICES TO BE MADE IN PICKING WHICH OF THE NEXT MODULE TO WORK ON.

WORK COULD START WITH THE INPUT SIDE OR THE OUTPUT SIDE. THERE IS NO FIRM RULE ON WHICH WORKS OUT BEST.

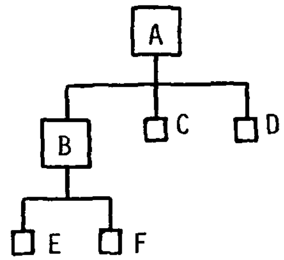
UNIT TESTING

TOP DOWN CODING AND TESTING

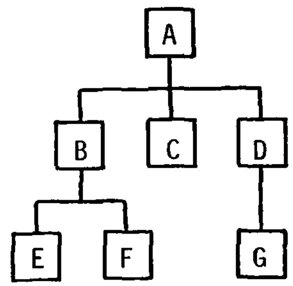
- CODE AND TEST THE TOP LEVEL MODULE USING STUBS FOR THE MODULES THAT ARE CALLED



- CODE NEXT LEVEL MODULES ONE (OR A FEW) AT A TIME AND TEST USING THE TOP LEVEL MODULE AS DRIVER AND STUBS FOR THE STILL LOWER LEVEL MODULES



- CONTINUE IN THIS WAY UNTIL ALL MODULES ARE CODED AND TESTED



POINT OUT THE VARIOUS POSSIBILITIES FOR STUBS.

AN EXAMPLE OF THE LAST TYPE OF STUB IS A NAVIGATION SYSTEM INTERFACE STUB THAT SIMULATES THE FLIGHT OF AN AIRCRAFT, SUPPLYING REALISTIC NAVIGATION DATA FOR THE REST OF THE SOFTWARE TO WORK ON.

EMPHASIZE THE NEED TO PLAN FOR (AND ALLOCATE TIME FOR) CONSTRUCTING THE NECESSARY STUBS. IF ELABORATE STUBS ARE REQUIRED THEY COULD REQUIRE A NON-TRIVIAL EFFORT.

UNIT TESTING

TOP DOWN CODING AND TESTING

STUBS

- STUBS REPRESENT AS YET UNIMPLEMENTED PARTS OF THE SYSTEM

- STUBS MAY
  - DO NOTHING (SIMPLY RETURN WHEN CALLED)
  - PRINT THEIR NAME AND MAYBE THEIR ARGUMENTS TO ALLOW TRACING PROGRAM FLOW
  - PRODUCE FIXED RESULTS
  - PRODUCE RANDOM RESULTS
  - PRODUCE RESULTS FROM A TABLE OR FILE

    RESULTS PRODUCED ALLOW OTHER MODULES TO BE TESTED WITH REALISTIC DATA
  - BE A SIMPLER SIMULATION OF THE FUNCTIONS OF THE MODULE

- THE TYPES AND FUNCTIONS OF THE STUBS SHOULD BE SPECIFIED IN THE UNIT LEVEL TEST PLAN

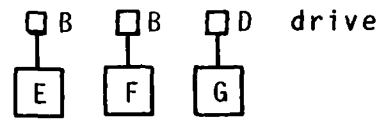
INSTRUCTOR NOTES

IN BOTTOM UP CODING AND TESTING THERE IS A CHOICE OF WHICH PATH TO BUILD UP FIRST -- THE INPUT PATH OR THE OUTPUT PATH. AGAIN, THERE IS NO FIRM RULE TO DECIDE.

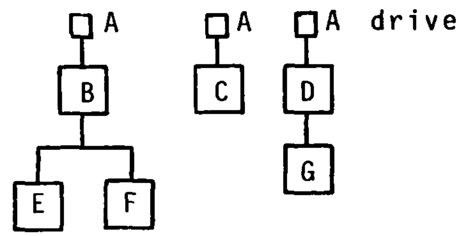
UNIT TESTING

BOTTOM UP CODING AND TESTING

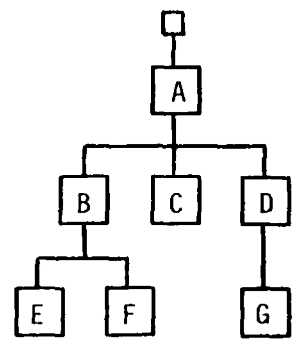
- OPPOSED TO TOP DOWN CODING AND TESTING IS THE BOTTOM UP APPROACH IN WHICH BOTTOM LEVEL MODULES ARE CODED AND TESTED WITH DRIVERS TO SUPPLY DATA AND EVALUATE RESULTS



- LOWER LEVEL MODULES ARE THEN INTEGRATED WITH A HIGHER LEVEL MODULE WITH ANOTHER DRIVER REPRESENTING THE STILL HIGHER LEVEL MODULE



- CONTINUE IN THIS WAY UNTIL ALL MODULES ARE CODED AND TESTED.



INSTRUCTOR NOTES

IN GENERAL THE BULK OF CODING AND TESTING IN A SYSTEM SHOULD BE TOP-DOWN.

IT MAY BE NECESSARY TO TEST SOME PARTS OF THE SYSTEM IN BOTTOM-UP FASHION, FOR EXAMPLE, I/O DRIVERS MAY HAVE TO BE WRITTEN AND TESTED BEFORE OTHER TOP-DOWN TESTING CAN BE DONE.

GENERALLY A LARGE SYSTEM IS WRITTEN AS A COLLECTION OF SUBSYSTEMS. EACH SUBSYSTEM IS TYPICALLY THE RESPONSIBILITY OF A SEPARATE PROGRAMMING TEAM. THE SUBSYSTEMS ARE USUALLY SEPARATELY WRITTEN AND THEN INTEGRATED TOGETHER (A BOTTOM-UP APPROACH). WITH IN A SUBSYSTEM IMPLEMENTATION SHOULD BE PRIMARILY TOP-DOWN. THERE IS CONSIDERABLE ADVANTAGE TO DOING A MEASURE OF TOP-DOWN INTEGRATION AMONG SEPARATE SUBSYSTEMS. THIS ALLOWS EARLY CHECKOUT OF SUBSYSTEM INTERFACES (THESE ARE PRIME TROUBLE SPOTS).

THIS TOPIC OFTEN PROVOKES A DISCUSSION. THAT IS A GOOD WAY TO END UP THIS SECTION IF THERE IS TIME.

UNIT TESTING

TOP DOWN TESTING ADVANTAGES

- TOP LEVELS ARE TESTED MOST THOROUGHLY -- WHICH IS GOOD SINCE THEY USUALLY ARE THE MOST CRITICAL
- EACH MODULE IS TESTED IN THE ACTUAL CONTEXT IN WHICH IT WILL BE USED
- STUBS ARE GENERALLY EASIER TO WRITE THAN DRIVERS
- INTEGRATION IS DONE IN SMALLER STEPS
- IN LOOKING AT A WHOLE SYSTEM THERE MAY BE SOME REASON TO CODE AND TEST SOME PARTS MOSTLY BOTTOM UP
- WITHIN A SUBSYSTEM HOWEVER IN MOST CASES CODING SHOULD BE TOP DOWN.

INSTRUCTOR NOTES

EXERCISE

WRITE A UNIT TEST PLAN FOR THE MERGE EXERCISE OF SECTION 2.

INSTRUCTOR NOTES

Section 5
REVIEW AND WRAP-UP

INSTRUCTOR NOTES

OUTLINE

1. INTRODUCTION
2. STRUCTURED PROGRAMMING
3. CODING STYLE
4. ENSURING RELIABILITY
5. **REVIEW AND WRAP-UP**

AD-A143 501 ADA (TRADEMARK) TRAINING CURRICULUM PROGRAMMING METHODOLOGY M203 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA JUL 84 DAAB07-83-C-K514

7/8

UNCLASSIFIED F/G 5/9 NL

END

Cont

1.0 2.8 2.5 3.2 2.2 3.6 2.0 4.0 1.1 1.8 1.25 1.4 1.6

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

INSTRUCTOR NOTES

HE FORMULATED THIS AS A RESULT OF COMPILING REFERRED PAPERS FOR THE QUOTED REFERENCE.

REVIEW

"THERE IS NO FIXED SET OF RULES ACCORDING TO WHICH CLEAR, UNDERSTANDABLE AND PROVABLE PROGRAMS CAN BE CONSTRUCTED."

<u>"GUEST EDITIOR'S OVERVIEW"</u>

P.J. DENNING, ACM COMPUTING SURVEYS, DEC. 1974.

INSTRUCTOR NOTES

- AN ISSUE DEVOTED ENTIRELY TO STRUCTURED PROGRAMMING
- A CLASSIC
- INCOMPARABLE - NOTHING EXISTS TODAY THAT CAN COMPARE
- A GOOD GUIDE TO WRITING IN GENERAL

BIBLIOGRAPHY

- ACM COMPUTING SURVEYS, DECEMBER 1974
- DAHL, O. DIJKSTRA, E.; AND HOARE, C. "STRUCTURED PROGRAMMING," ACADEMIC PRESS, NEW YORK, 1972
- KERNIGHAN, B.W. AND PLAUGER, P.J., "THE ELEMENTS OF PROGRAMMING STYLE," McGRAW-HILL, NEW YORK, 1974
- STRUNK, W. AND WHITE, E.B., "THE ELEMENTS OF STYLE," MACMILLAN, NEW YORK, 1959

INSTRUCTOR NOTES

REVIEW

- MUST BE ABLE TO SEE PART OF THE PROGRAMMERS THOUGHT PROCESSES, STARTING FROM THE ORIGINAL (VERY ABSTRACT) AND PROCEEDING TO THE END VIA A CLEARLY PRESENTED SEQUENCE OF TRANFORMATIONS AND REFINEMENTS

- IF THE ABOVE IS NOT POSSIBLE, THE CODE IS OBSCURE

INSTRUCTOR NOTES

POINT OUT THAT THESE ARE EQUALLY IMPORTANT.

REVIEW

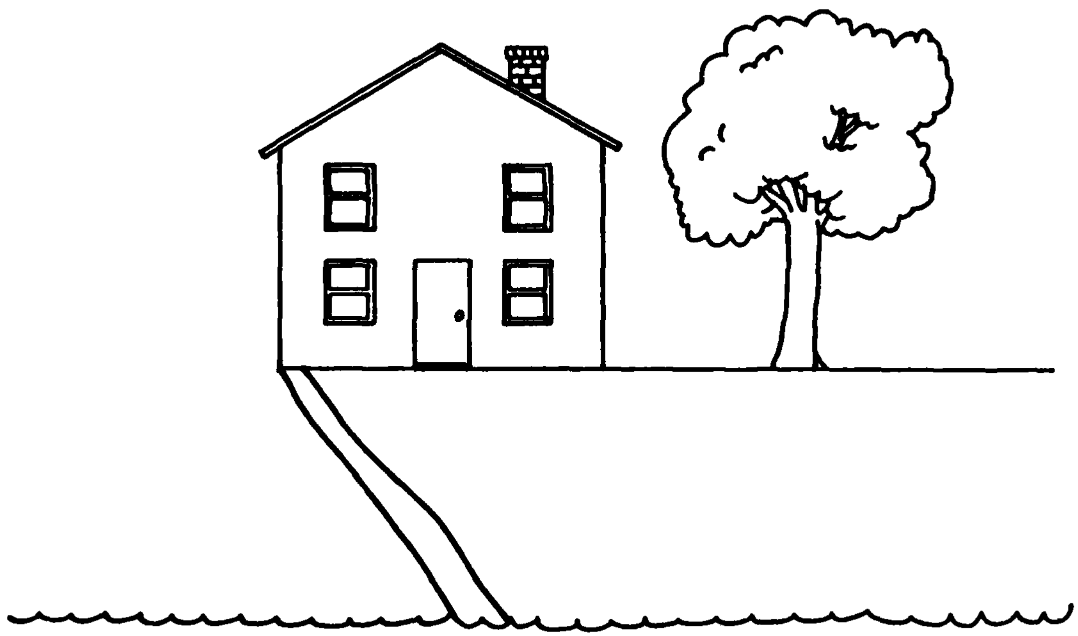
- PUT YOURSELF IN THE SHOES OF OTHERS
- DOCUMENT WELL
- VERIFY AND TEST
- PLAN
- CONSIDER WHAT MIGHT GO WRONG

P.J. BROWN, "PROGRAMMING AND DOCUMENTING SOFTWARE PROJECTS," ACM COMPUTING SURVEYS, DECEMBER 1974.

INSTRUCTOR NOTES

THIS FOIL AND THE FOLLOWING TWO (2) FOILS ARE SELF EXPLANATORY.

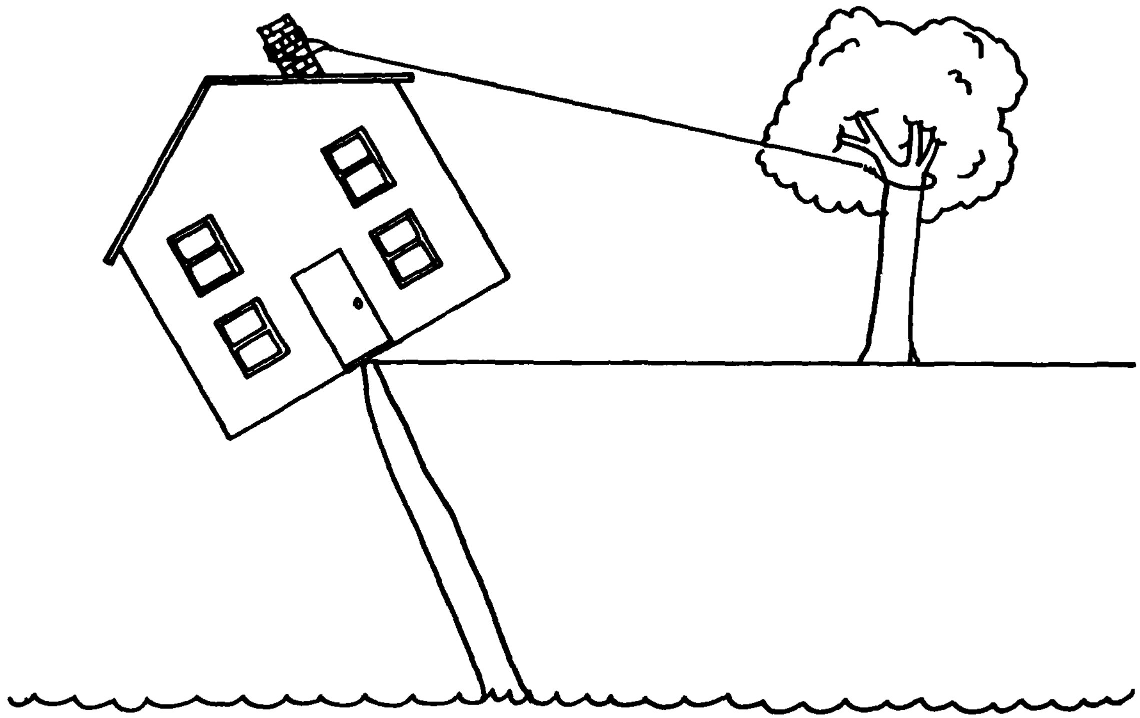
IF THE INITIAL DESIGN IS WRONG



P.J. BROWN, "PROGRAMMING AND DOCUMENTING SOFTWARE PROJECTS," ACM COMPUTING SURVEYS, DECEMBER 1974.

INSTRUCTOR NOTES

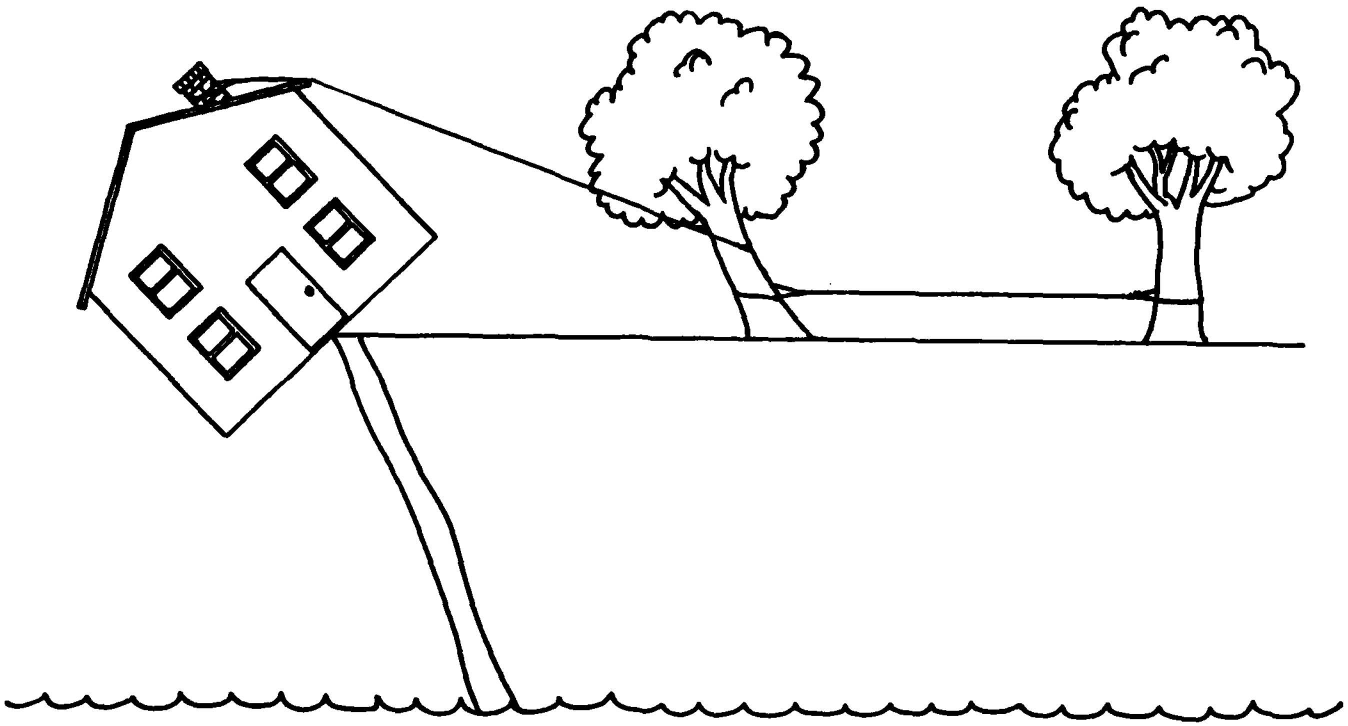
IT CAN ALWAYS BE FRIGGED ...



P.J. BROWN, "PROGRAMMING AND DOCUMENTING SOFTWARE PROJECTS," ACM COMPUTING SURVEYS, DECEMBER 1974.

INSTRUCTOR NOTES

... THOUGH THIS MAY LEAD TO FURTHER DIFFICULTIES



P.J. BROWN, "PROGRAMMING AND DOCUMENTING SOFTWARE PROJECTS," ACM COMPUTING SURVEYS, DECEMBER 1974.

INSTRUCTOR NOTES

IS THIS ADA?

KNUTH

"... MY DREAM IS THAT BY 1984 WE WILL SEE A CONSENSUS DEVELOPING FOR A REALLY GOOD PROGRAMMING LANGUAGE. I'M GUESSING THAT PEOPLE WILL BECOME SO DISENCHANTED WITH THE LANGUAGES THEY ARE NOW USING THAT THIS NEW LANGUAGE, UTOPIA 84, WILL HAVE A CHANCE TO TAKE OVER..."

"STRUCTURED PROGRAMMING WITH GO TO STATEMENTS," ACM COMPUTING SURVEYS, DEC. 1974

END

FILMED

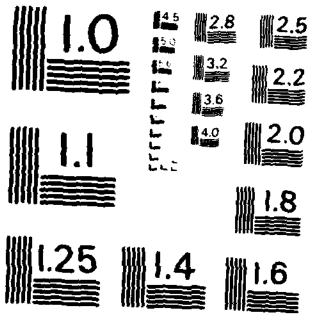
9-84

DTIC

AD-A143 581 ADA (TRADEMARK) TRAINING CURRICULUM PROGRAMMING METHODOLOGY M203 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA JUL 84 DAAB07-83-C-K514 8/8

UNCLASSIFIED F/G 5/9 NL

END
DATE
FILMED
11-84
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

SUPPLEMENTARY

INFORMATION

AD-A143 581

DEPARTMENT OF THE ARMY
HEADQUARTERS US ARMY COMMUNICATIONS-ELECTRONICS COMMAND
AND FORT MONMOUTH
FORT MONMOUTH, NEW JERSEY 07703

REPLY TO
ATTENTION OF:

15 OCT 1984

Center for Tactical Computer Systems

Ms. Madeline Crumbacker
Defense Tactical Information Center
Cameron Station
Alexandria, Virginia 22314

Dear Ms. Crumbacker:

As per phone conversation with Ms. Andrea Cappellini, CENTACS on 11 October 1984, a copyright statement has been omitted on documents sent to DTIC and NTIS. Enclosed please find the copyright statement (Encl 1) that must appear in the enclosed list of document (Encl 2). If you have any questions, please contact Ms. Cappellini at 201-544-4280.

Sincerely,

Dennis J. Turner
for JAMES E. SCHELL
Director, CENTACS

DATE
ILME